Rectilinear minimum link paths in two and higher dimensions

Mikko Sysikaski

Helsinki December 15, 2018

Master's thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Computer Science

Tekijä — Författare — Author
Mikko Sysikaski

Työn nimi — Arbetets titel — Title

Rectilinear minimum link paths in two and higher dimensions

Ohjaajat — Handledare — Supervisors
Antti Laaksonen and Jyrki Kivinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	December 15, 2018	65 pages + 1 appendix

Tiivistelmä — Referat — Abstract

The thesis discusses algorithms for the minimum link path problem, which is a well known geometric path finding problem. The goal is to find a path that does the minimum number of turns amidst obstacles in a continuous space. We focus on the most classical variant, the *rectilinear* minimum link path problem, where the path and the obstacles are restricted to the directions of the coordinate axes.

We study the rectilinear minimum link path problem in the plane and in the three-dimensional space, as well as in higher dimensional domains. We present several new algorithms for solving the problem in domains of varying dimension. For the planar case we develop a simple method that has the optimal $O(n \log n)$ time complexity. For three-dimensional domains we present a new algorithm with running time $O(n^2 \log^2 n)$, which is an improvement over the best previously known result $O(n^{2.5} \log n)$. The algorithm can also be generalized to higher dimensions, leading to an $O(n^{D-1} \log^{D-1} n)$ time algorithm in D-dimensional domains.

We describe the new algorithms as well as the data structures used. The algorithms work by maintaining a reachable region that is gradually expanded to form a shortest path map from the starting point. The algorithms rely on several efficient data structures: the reachable region is tracked by using a simple recursive space decomposition, and the region is expanded by a sweep plane method that uses a multidimensional segment tree.

ACM Computing Classification System (CCS):
Theory of computation → Design and analysis of algorithms → Graph algorithms analysis → Shortest paths
Theory of computation → Randomness, geometry and discrete structures → Computational geometry

Avainsanat — Nyckelord — Keywords
algorithms, computational geometry, route planning

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Algorithms, Data Analytics and Machine Learning subprogramme

Contents

Appendices

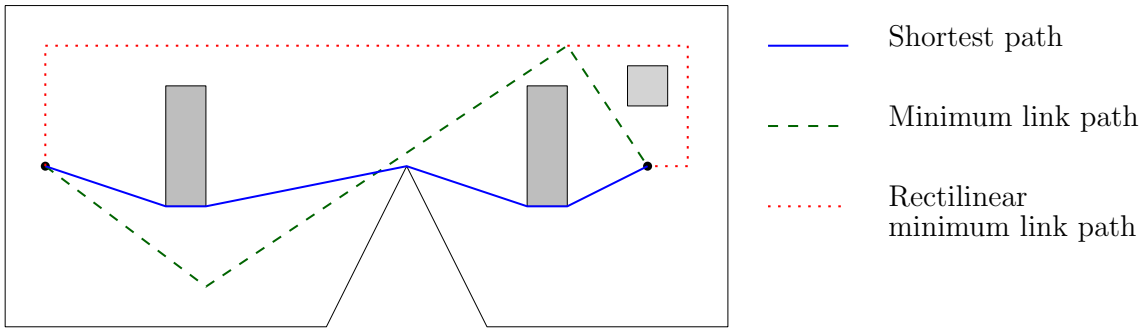


Figure 1: There are many variants of the geometric path finding problem, producing different kinds of paths.

1 Introduction

Various kinds of path finding problems are a common topic in algorithmics [14]. The regular shortest path problem asks for the shortest path between two nodes in a graph. The geometric version of the problem asks for the shortest path in a continuous space, such as a plane with polygonal obstacles. The typical goal is to minimize the Euclidean length of the path, but for some applications other distance metrics are more relevant. For example a robot might move quickly in a straight line but take a lot of time to turn, so a long but simple route can be preferable to a shorter but more complex path.

The topic of this thesis is finding geometric paths that minimize the number of turns. Consider a path consisting of a finite number of straight line segments, also called *links*. The *link distance* of the path is the number of segments. A *minimum link path* between two points is a path that minimizes the link distance. Figure 1 shows an example of two kinds of minimum link paths as well as the regular shortest path.

Finding minimum link paths is a well studied algorithmic problem [3, 4, 6, 8, 9, 16, 21]. The basic idea employed by most of the known algorithms is a simple extension of breadth-first search: We first mark all the points that have link distance 1 from the starting point, then points with link distance 2, and continue until the desired end point is marked. The path is then formed by tracing back the markings starting from the end point. As the search happens in continuous space, the tricky part of the algorithms is defining the appropriate data structures to store and update the marked region.

In this thesis we focus on the *rectilinear* variant of the problem. In the rectilinear problem the obstacles as well as the links of the path are restricted to the coordinate

axes' directions. The problem can also be defined in a three-dimensional domain as well as higher dimensions.

Following a typical convention, we assume that the domain is given as a list of n rectangular obstacle faces [19]. In the planar case the obstacle faces are line segments, and in a D-dimensional domain they are $(D-1)$-dimensional hyperrectangles. We use the terms "obstacle" and "obstacle face" interchangeably to refer to an obstacle face in the input. Each obstacle is also associated with a normal vector that points towards the free space.

We present new algorithms for both the two dimensional case and the higher dimensional variants. For the planar case we design an algorithm that solves the problem in optimal $O(n \log n)$ time and $O(n)$ space. The new algorithm is simpler than the previously known solutions and has the same time and space complexity. We also introduce a new $O(n^2 \log^2 n)$ time algorithm for the three-dimensional case, which is a significant improvement over the best previously known result $O(n^{2.5} \log n)$ [16]. Finally, we extend the solution to work in any D-dimensional rectilinear domain with $O(n^{D-1} \log^{D-1} n)$ time complexity.

1.1 Background and new results

Computing rectilinear minimum link paths in the plane is an old and well studied problem. The problem was first presented in 1968 by Mikami and Tabuchi [13], who also gave an algorithm with $O(n^2)$ running time. Sato, Sakanaka and Ohtsuki [17] presented an algorithm with $O(n \log n)$ time and $O(n)$ space complexity. A similar result was achieved independently by Das and Narasimhan [3]. These algorithms are asymptotically optimal, as the problem can be shown to be as hard as sorting [3] and thus having lower bound $O(n \log n)$.

Several algorithms have also been developed for the higher dimensional rectilinear problem. De Berg et al. [6] gave an $O(n^D \log n)$ time algorithm for computing a minimum link path in a D-dimensional domain. For the three-dimensional case, an $O(n^3)$ algorithm was developed by Mikami and Tabuchi [13]. Fitch, Butler and Rus [8] gave an algorithm which improves the performance in many cases, but has the same $O(n^3)$ worst case running time. This was further improved by Wagner, Drysdale and Stein [21], who gave an algorithm with worst case $O(n^{2.5} \log n)$ time complexity.

In this thesis we focus on new results for the rectilinear minimum link path problem,

which are original research of the author. The following results have been published in peer-reviewed journals and conferences.

- A simplified $O(n \log n)$ algorithm for the two-dimensional case [15]. The article was written with the co-authors Valentin Polishchuk and Joseph S. B. Mitchell. The main ideas of the algorithm were invented by the author, though many details were worked out with Polishchuk. The contribution of Mitchell was mainly for other results not discussed in this thesis, such as the approximation algorithm for the non-rectilinear minimum link path problem.
- A new algorithm for the three-dimensional minimum link path problem, improving the running time from the previous best result of $O(n^{2.5} \log n)$ to $O(n^2 \log^2 n)$ [16]. The publication was written with the co-author Valentin Polishchuk. The main ideas were invented by the author, but many details and proofs of the running time were worked out together with Polishchuk.

The following items are new contributions in this thesis.

- Detailed description of the algorithm for the three-dimensional minimum link path problem and the data structures used in the algorithm.
- Simplification of the three-dimensional minimum link algorithm by using the unified segment tree by Wagner [20].
- Extension of the space decomposition used in the algorithms to higher than three dimensions.
- Extension of the unified segment tree to higher than two dimensions and a method for clearing a hyperrectangle in the tree. The original paper by Wagner [20] mentions the possibility of extending the structure to higher dimensions but does not provide any details.
- An algorithm for the minimum link path problem in higher dimensions. We show that the rectilinear minimum link path problem in D-dimensional domains can be solved in $O(n^{D-1} \log^{D-1} n)$ time by a generalization of the algorithm for the three-dimensional case.

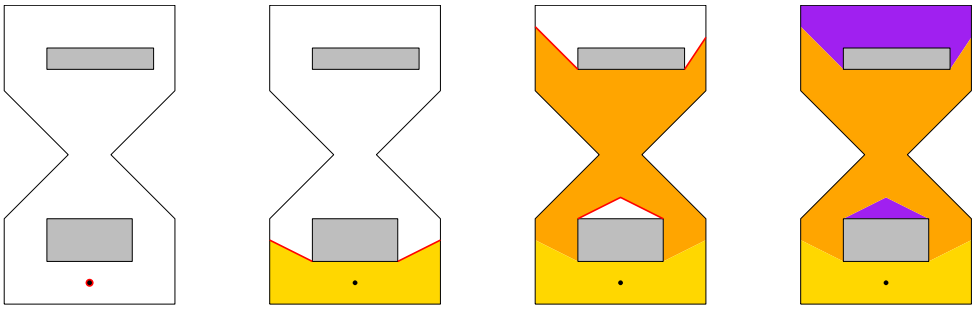


Figure 2: Four steps of staged illumination from a single starting point. The new areas are illuminated through the red boundaries on each step.

1.2 Overview of the algorithms

A majority of the known algorithms for the minimum link path problem share a common basic idea [9, Sections 26.4, 27.3]: Let $\mathcal{A}$ be the region not blocked by obstacles, and let $\mathbf{s} \in \mathcal{A}$ and $\mathbf{t} \in \mathcal{A}$ be the start point and the end point of the path. The region $\mathcal{A}$ is also called the *free space*. Let $\textsc{reach}(k) \subseteq \mathcal{A}$ be the set of points reachable by at most k links from $\mathbf{s}$. We start by defining $\textsc{reach}(0) = \{\mathbf{s}\}$, and iteratively compute $\textsc{reach}(k+1)$ based on $\textsc{reach}(k)$. The iteration continues until we find k such that $\mathbf{t} \in \textsc{reach}(k)$. This iteration is often called the *staged illumination*, because the computation of $\textsc{reach}(k+1)$ can be illustrated as placing light sources on the boundaries of $\textsc{reach}(k)$ and observing the region that gets lit. Figure 2 shows an example of the staged illumination process.

The illumination process closely resembles breadth-first search, though with a key difference that the search is done in a continuous space rather than in a graph. A nontrivial part of the minimum link path algorithms is maintaining the illuminated area in a data structure that allows efficient computation of $\textsc{reach}(k+1)$ from $\textsc{reach}(k)$.

Representing the illuminated area can be simplified by first splitting the free space into rectangles. This is analogous to how various algorithms in computational geometry construct a triangulation of the input region as the first step [9]. In Section 2 we present an algorithm to construct a decomposition suitable for the minimum link path algorithms.

Because the possible directions of the links of the path are limited, we can use *sweep line algorithms* to efficiently compute $\textsc{reach}(k+1)$ from $\textsc{reach}(k)$. Sweep line algorithms are a generic technique used for a wide range of computational geometry problems [2, Section 33.2]. The idea can be visualized by a moving line

that crosses over the domain and stops at some discrete *events*. As the sweep line progresses, we gradually update the solution to take into account the parts of the domain encountered in the events. The sweep line method allows us to transform the two-dimensional illumination problem into a series of easier one-dimensional problems.

During the sweeps we need to maintain the relationship between the sweep line and the obstacles as well as the region $\textsc{reach}(k)$. We need to maintain the intersection of the sweep line with the illuminated area, which requires efficient handling of intervals of real numbers. A *segment tree* is a generic data structure for working with intervals. It is flexible enough to support all the operations required by the staged illumination. Segment trees are described in Section 3.

We use the aforementioned concepts to build a simple algorithm for finding minimum link paths in Section 4. The ideas of the two-dimensional solution can also be generalized to work in higher dimensions. Many details need to be taken care of, but the basic ideas are relatively straightforward to transfer to more than two dimensions. The free space is decomposed into cuboids instead of rectangles, and the sweep line is replaced by a *sweep plane*. The segment tree can also be generalized into a multidimensional structure. These elements are combined into an algorithm for finding minimum link paths in three and higher dimensional domains in Section 5.

2 Space decomposition

The first step of many minimum link path algorithms is decomposing the free space into simple primitives [3, 10, 15]. The decomposition plays two roles during the staged illumination: it is used to track the illuminated area, and to guide the sweep line algorithms that illuminate new areas.

In the rectilinear minimum link path problem the obstacle edges are oriented according to the coordinate axes. This allows us to decompose the free space into axis-aligned rectangles, which is a convenient form for finding rectilinear paths. The rectangles in the decomposition are also called *cells*. We also store a list of links to neighboring cells and obstacle faces for each cell of the decomposition. Thus the decomposition can be seen as a graph with cells as vertices and an edge between each pair of adjacent cells. Recall that n is the total number of obstacle faces in the input. We will present a method that decomposes a two-dimensional rectilinear domain into $O(n)$ cells such the total number of links between cells is $O(n)$.

Figure 3: The horizontal decomposition and the vertical decomposition of the same domain. Each cell touches at least one obstacle corner and each obstacle corner touches at most two cells, so both decompositions have size $O(n)$.

A similar structure can also be used for computing minimum link paths in three-dimensional domains. In the three-dimensional rectilinear case each obstacle face is an axis-aligned rectangle in the three-dimensional space. The free space of a three-dimensional domain can be decomposed into a graph with $O(n^2)$ cuboids as vertices and $O(n^2)$ edges. This decomposition algorithm can also be extended to higher dimensions: we will show how to decompose a D-dimensional domain into $O(n^{D-1})$ hyperrectangles with $O(n^{D-1})$ links between them.

2.1 Planar decomposition

A rectilinear domain can be decomposed into rectangles by extending each horizontal obstacle edge in both directions until the sides hit a vertical obstacle edge [10]. This defines the *horizontal decomposition* of the domain, denoted DEC_x. The *vertical decomposition* DEC_y is defined correspondingly by extending all the vertical edges until they hit a horizontal obstacle. Figure 3 shows an example of the two decompositions. We focus on DEC_x below, as DEC_y has all the same properties with coordinate axes swapped.

It is easy to see that each cell of DEC_x touches at least one obstacle vertex. Since each vertex can touch at most two rectangles, the size of the decomposition is $O(n)$. Each pair of touching rectangles also shares a common obstacle corner, so the number of links is also $O(n)$.

Each rectangle a in the decomposition is defined by its x-range $x(a)$ and the y-range $y(a)$, which we assume to be half-open intervals. We refer to lower and upper bound of each range r by r_{start} and r_{end}.

DEC_x can be constructed by a line sweep algorithm [3]. We sweep over the domain by a horizontal line moving from y coordinate $-\infty$ (down) to ∞ (up). During the sweep we maintain the intersection of the free space $\mathcal{A}$ and the sweep line. The

intersection consists of non-overlapping intervals, which are used as the x-ranges of the constructed rectangles. The rectangles touching the sweep line are stored in a binary tree during the sweep. When a rectangle a is added to the tree, $y(a)_{end}$ is initially left undefined because we don't know how tall the rectangle will be until a is removed from the tree.

Algorithm 2.1 shows the sweep line process. The sweep stops at each horizontal obstacle edge. The normal of the edge points towards the free space, which is either below or above. If the normal points down, the x-range of the obstacle is fully contained in one of the x-ranges of the rectangles in the search tree. If the normal points up, the obstacle range does not intersect any x-ranges in the tree, but the left and right endpoints may touch some of the rectangles. In both cases the rectangles touching the obstacle edge are removed from the search tree, and $O(1)$ new rectangles are inserted to maintain the sweep line state.

In addition to the cells, we also want to compute the links between all the adjacent pairs. It is straightforward to extend Algorithm 2.1 to compute the links by inserting links between the cells that are removed and inserted in the binary search tree during each event. The following lemma gives a bound on the complexity of the algorithm.

Lemma 2.1. *Algorithm 2.1 works in $O(n \log n)$ time and $O(n)$ space.*

Proof. The algorithm sorts all the horizontal edges in $O(n \log n)$ time and iterates over them in $O(n)$ steps. On each step we create $O(1)$ new rectangles and perform $O(1)$ operations in the binary search tree. The search tree is a balanced binary tree, so each tree operation is carried out in $O(\log n)$ time. Thus the total complexity of the algorithm is $O(n \log n)$ and we only need $O(n)$ storage for the edge list and the binary search tree as well as for storing the output. □

2.2 Three-dimensional cuboid decomposition

Like a two-dimensional domain can be decomposed into rectangles, a three-dimensional rectilinear domain can be decomposed into cuboids [16, 21]. A cuboid is a three-dimensional hyperrectangle. Decomposition into cuboids is used in the algorithm for three-dimensional minimum link paths in Section 5. The running time of the path finding algorithm depends on the size of the decomposition, so it is desirable to find as small a decomposition as possible.

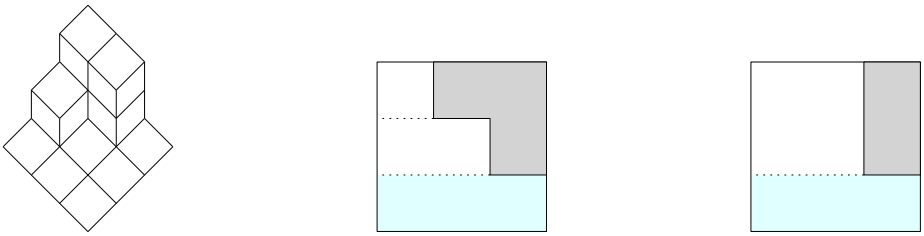
We present a simple decomposition algorithm that uses Algorithm 2.1 as a sub-

Algorithm 2.1 Decompose the free space into DEC_x.

Input: Set of obstacle line segments E.

Output: Decomposition DEC_x of the region defined by E.

```
T ← Empty binary search tree.
Sort E by the y coordinates of the edges.
R ← Empty list of cells.
for all e ∈ E do
    if Normal of e points down then
        ▷ Exactly one x-interval in T intersects x(e).
        v ← Lookup T for interval touching x(e).
        Remove v from T.
        y(v)_end ← e_y.
        Insert v into R.
        ▷ Loop over remaining parts of the entry removed from T.
        for all u ← x(v)\x(e) do
            a ← New rectangle with x-range u and starting y-coordinate e_y.
            Insert a into T.
        end for
    else                                          ▷ Normal of e points up.
        a ← New rectangle with x-range x(e) and starting y-coordinate e_y.
        for all v ∈ lookup T for intervals touching x(e) do
            Remove v from T.
            y(v)_end ← e_y.
            Insert v into R.
            x(a) ← x(a) ∪ x(v).
        end for
        Insert a into T.
    end if
end for
return R.
```



(a) Three-dimensional domain (b) Cross section of the lower level (c) Cross section of the upper level

Figure 4: A three-dimensional domain is decomposed into cuboids by taking the horizontal decomposition of each cross section. The highlighted cell is shared between the two cross sections, so the cuboids on the two levels are merged.

routine [16]. The algorithm produces a decomposition with $O(n^2)$ cells and $O(n^2)$ links, where n is the number of obstacle rectangles. Earlier work on minimum link path algorithms has used binary space partitioning to reduce the number of cells to $O(n^{1.5})$ [7, 21]. However the number of links in a binary space partitioning based decomposition can be as high as $O(n^{2.5})$, so we opt for a higher number of cells to reduce the number of links.

The decomposition is created by a sweep plane algorithm, which sweeps through the domain with a plane perpendicular to z-axis. During the sweep we maintain the intersection of the sweep plane and the domain $\mathcal{A}$. The intersection of the domain and a plane is called a *cross section* of the domain.

Algorithm 2.2 constructs the cells of the decomposition. The algorithm forms the planar decomposition of each unique cross section by calling Algorithm 2.1, and extends the xy-rectangles in z-direction so that they fill the whole free space. Any common rectangles shared by consecutive cross sections are merged into a single larger cuboid. The merging is done by maintaining a binary search tree of the cells of the previous decomposition. Figure 4 shows how the three-dimensional decomposition is created using a series of planar decompositions.

Lemma 2.2. *Algorithm 2.2 has running time $O(n^2 \log n)$.*

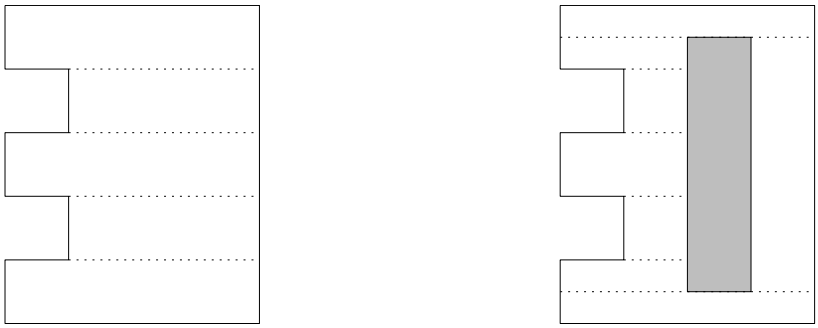
Proof. We loop through $O(n)$ unique z-coordinates. For each z we construct a set of obstacles in $O(n)$ time and a planar decomposition in $O(n \log n)$ time. The map M is implemented as a binary search tree, so each update and lookup can be performed in $O(\log n)$ time. For each z we perform $O(n)$ map updates, so the total time complexity is $O(n^2 \log n)$. □

Algorithm 2.2 Decompose the three-dimensional free space into cuboids.

Input: Set of rectangular obstacle faces E.

Output: Three-dimensional decomposition of the domain defined by E.

```
Z ← All z-coordinates of the obstacles in E.
Sort Z in increasing order.
R ← Empty list of result cells.
M ← Empty binary search tree mapping rectangles to indices in R.
for all z ∈ Z do
    E_z ← {e | e ∈ E, z(e) ∋ z}.
    T ← planar decomposition for E_z using Algorithm 2.1.
    for all m ∈ M\T do
        c ← R[M[m]].
        z(c)_end ← z.
        Remove m from M.
    end for
    for all t ∈ T\M do
        c ← New cell with xy-bounds t and starting z-coordinate z.
        Insert c into R.
        M[t] ← index of c in R.
    end for
end for
return R.
```



(a) Original domain (b) Domain after adding an obstacle

Figure 5: Adding or removing a single obstacle can change all $O(n)$ cells of a horizontal decomposition. However the intersection of the domain with any horizontal line is changed at only $O(1)$ points. Because there are $O(n)$ unique cross sections, the number of overlapping pairs of rectangles in the old and the new domain is $O(n)$.

Next we show how to compute the links between the cuboids generated by Algorithm 2.2. The following lemma shows how the changes to the cross section affect the number of links. Figure 5 illustrates the case where two adjacent cross sections differ by a single obstacle rectangle.

Lemma 2.3. *Consider two obstacle sets A and B in the two-dimensional plane. Let n be the total number of obstacle edges in A and B, and k be the number of edges that are present in exactly one of the sets. The number of overlapping rectangles in* $\text{DEC}_x(A)$ *and* $\text{DEC}_x(B)$ *is* $O(nk)$.

Proof. We divide the intersecting rectangle pairs into two groups: pairs with identical x-range and pairs with different x-ranges. Let a and b be two rectangles with $x(a) = x(b)$. If $a \cap b \neq \varnothing$, then either $y(a)_{start} \in y(b)$ or $y(b)_{start} \in y(a)$. Since the starting y-coordinate of any rectangle can be inside only a single rectangle with the same x-range, the number of overlapping pairs with equal x-range is $O(n)$.

To count the overlapping pairs with different x-ranges, consider how the sweep line of Algorithm 2.1 advances in the domains A and B. Recall that the sweep line is a horizontal line moving in y-direction.

The intersection of the sweep line and the free space is a sequence of disjoint intervals for both A and B. Since there are $O(k)$ changes to the set of obstacles, the edit distance between the two interval sets is $O(k)$ during each point of the sweep. Thus each y coordinate contributes $O(k)$ pairs to the overlap. Since the number of y coordinates where the domain changes is $O(n)$, the total number of overlapping pairs

with different x-ranges is $O(nk)$.

The total number of overlapping pairs is the sum of the pairs with equal x-ranges and the pairs with different x-ranges $O(n + nk) = O(nk)$. □

Lemma 2.4. *The number of pairs of adjacent cells in the decomposition produced by Algorithm 2.2 is* $O(n^2)$.

Proof. Since the number of links produced by Algorithm 2.1 is $O(n)$, any cross section has $O(n)$ links in x and y directions. Thus the total number of x and y links is $O(n^2)$.

Let k_z be the number of obstacles whose z-range has z as either the lower bound or the upper bound. Since the total number of obstacles is n, the sum of k_z values over all z-coordinates is $O(n)$. By Lemma 2.3 the number of links between two adjacent layers is $O(nk_z)$, so the total number of z-links is $\sum_z O(nk_z) = O(n^2)$. □

To generate the links, notice that the links in x and y directions can be simply extracted from the planar decompositions of each cross section. Some links may be generated multiple times but the number of generated links is still $O(n^2)$, so we can find and remove the duplicates by sorting the links in $O(n^2 \log n)$ time. What remains is computing the links in z direction. We compute the links between each pair of adjacent cross sections separately.

Let A and B be the sets of rectangles that are different on the horizontal decompositions of two adjacent cross sections. Finding the z-links in the three-dimensional decomposition is equivalent to finding the set of overlapping pairs between A and B. The pairs are found by running *another* line sweep algorithm after both A and B have been created by Algorithm 2.1.

Algorithm 2.3 shows the computation of the overlapping pairs. The algorithm sweeps through the domain by a horizontal line. During the sweep we maintain two binary search trees T_A and T_B, containing the rectangles of A and B touching the sweep line respectively. The binary search trees are ordered by the x-ranges of the rectangles. Note that the rectangles in each of the sets A and B are disjoint, so the x-ranges are disjoint as well.

Each time the sweep line arrives into a new rectangle $a \in A$, the tree T_B is queried to find all the rectangles touched by the bottom line of a. Similarly for each new rectangle $b \in B$ we query the tree T_A for rectangles touching the bottom line of b.

Algorithm 2.3 Find all the overlapping pairs in two sets of rectangles.

Input: Sets of disjoint rectangles A and B.

Output: All the overlapping pairs of elements in A and B.

```
E ← All the bottom and top edges of the rectangles in A and B.
Sort E by y-coordinates.
T_A ← Empty binary search tree.
T_B ← Empty binary search tree.
R ← Empty list of pairs.
for all e ∈ E do
    s ∈ {A, B} ← Group to which e belongs to.
    t ∈ {A, B} ← Group to which e does not belong to.
    r ← rectangle whose edge e is.
    if e is a bottom edge of r then
        for all u ∈ elements in T_t intersecting e do
            Insert (r, u) into R.
        end for
        Add r into T_s.
    else
        Remove r from T_s.
    end if
end for
return R.
```

Lemma 2.5. *Algorithm 2.3 runs in $O(n \log n + k)$ time, where n is the total size of A and B, and k is the number of overlapping pairs.*

Proof. First we sort all the top and bottom edges in the input in $O(n \log n)$ time. We then iterate over all the edges, performing three kinds of tree operations: add, remove and lookup.

Each add and remove operation takes $O(\log n)$ time in a balanced binary search tree. The lookup of the elements intersecting a given element is done by first finding the leftmost intersecting element in the tree and then scanning adjacent tree nodes until all the intersecting elements have been found. This requires time $O(\log n + u)$ where u is the number of overlapping elements.

Combining the time for sorting, tree additions, deletions and lookups, the total complexity is $O(n \log n + k)$. □

We are now ready to prove the main result of this section.

Theorem 1. *A three-dimensional rectilinear domain defined by n obstacle faces can be decomposed into $O(n^2)$ cuboids with $O(n^2)$ links between them in $O(n^2 \log n)$ time and $O(n^2)$ space [16].*

Proof. The cuboids can be constructed in $O(n^2 \log n)$ time according to Lemma 2.2. The total number of links between the cells is $k = O(n^2)$ according to Lemma 2.4. Computing the links between all the layers can then be done in $O(n^2 \log n + k) = O(n^2 \log n)$ time by Lemma 2.5. Thus the total time complexity is $O(n^2 \log n)$, and the total number of cuboids and links is $O(n^2)$.

Algorithm 2.2 and Algorithm 2.3 both only require $O(n)$ additional space for binary search trees and event sets, so the space complexity equals the size of the output $O(n^2)$. □

2.3 Higher dimensional decomposition

The ideas of the three-dimensional decomposition can be generalized to allow also decomposing higher dimensional rectilinear domains into hyperrectangles. A D-dimensional domain is defined by a set of $(D-1)$-dimensional hyperrectangle obstacle faces. The construction is done recursively so that the D-dimensional decomposition is created by solving a sequence of $(D-1)$-dimensional subproblems. The method is similar to how Algorithm 2.2 forms the three-dimensional decomposition from

a series of planar decompositions. We use the terms rectangle and hyperrectangle interchangeably when it is obvious that we are referring to multidimensional rectangles.

Algorithm 2.4 Decompose the free space into D-dimensional hyperrectangles.

Input: Set E of $(D-1)$-dimensional obstacle faces.
Output: Decomposition of the D-dimensional region defined by E.

```
if D = 2 then
    return Decomposition using Algorithm 2.1.
end if
Z ← All D-coordinates of the obstacles in E.
Sort Z in increasing order.
R ← Empty list of result cells.
M ← Empty binary search tree mapping (D − 1)-dimensional rectangles to indices
        in R.
for all z ∈ Z do
    E_z ← Obstacles of E whose D-range contains z.
    T ← (D − 1)-dimensional decomposition for E_z recursively.
    for all m ∈ M\T do
        c ← R[M[m]].
        Upper D-coordinate of c ← z.
        Remove m from M.
    end for
    for all t ∈ T\M do
        c ← New cell with the first D − 1 coordinates defined by t and starting
                D-coordinate z.
        Insert c into R.
        M[t] ← index of c in R.
    end for
end for
return R.
```

Algorithm 2.4 shows how the domain is split into hyperrectangles. The analysis is similar to the three-dimensional case, except that we now use induction on D.

Lemma 2.6. *The number of rectangles produced by Algorithm 2.4 is* $O(n^{D-1})$.

Proof. Proof by induction on D. Since the horizontal decomposition of a two-

dimensional domain has $O(n)$ cells, the claim is true for $D = 2$. For $D \geqslant 3$ we loop over the $O(n)$ unique values of the D-coordinate in the obstacles. For each cross section we compute a $(D-1)$-dimensional decomposition, whose size is $O(n^{D-2})$ by induction. The combination of the n cross sections has total size $O(n^{D-2}n) = O(n^{D-1})$. □

Lemma 2.7. *The running time of Algorithm 2.4 is $O(n^{D-1} \log n)$.*

Proof. Proof by induction on D. Since we use Algorithm 2.1 if $D = 2$, Lemma 2.1 proves the base case $D = 2$.

For $D \geqslant 3$ we solve $O(n)$ subproblems, each of which can be done in $O(n^{D-2} \log n)$ time by induction. The total time of solving all the subproblems is thus $O(n^{D-1} \log n)$.

We also maintain a mapping from $(D-1)$-dimensional hyperrectangles to result indices. The size of the mapping is bounded by the size of the $(D-1)$-dimensional decompositions, which have size $O(n^{D-2})$ by Lemma 2.6. Thus each map operation can be done in $O(\log n^{D-2}) = O(\log n)$ time. The number of map operations for each cross section is $O(n^{D-2})$, so the total time spent in the map operations is $O(nn^{D-2} \log n) = O(n^{D-1} \log n)$. □

Next we consider how to generate links for the adjacent pairs of a D-dimensional decomposition.

Lemma 2.8. *Consider two obstacle sets A and B that each define a D-dimensional domain. Let n be the total number of $(D-1)$-dimensional obstacle faces in A and B, and k be the number of faces that are present in exactly one of the sets. The number of pairs of overlapping D-dimensional rectangles in the decompositions of A and B is $O(n^{D-1}k)$.*

Proof. Proof by induction on D. The base case $D = 2$ is proven by Lemma 2.3.

For $D \geqslant 3$ consider how the sweeps of Algorithm 2.4 advance in the domains defined by A and B. Each of the sweeps stops at $O(n)$ points, producing a $(D-1)$-dimensional decomposition for the cross section. For any D-coordinate, the cross sections of the domains of A and B differ by $O(k)$ obstacles, so the number of overlapping pairs at a fixed D-coordinate is $O(n^{D-2}k)$ by induction.

The total number of overlapping pairs is bounded by the number of pairs counted in each of the $O(n)$ cross sections. This may count some pairs multiple times, but each

pair is counted at least once. Each cross section has $O(n^{D-2}k)$ overlapping pairs, so the total number of pairs has upper bound $O(n^{D-1}k)$. □

Lemma 2.9. *The number of links between adjacent pairs of cells produced by Algorithm 2.4 is* $O(n^{D-1})$.

Proof. Consider separately the links in direction D and in the other directions. For both cases, we count the number of links using induction on D. For the base case of the induction, the number of links in case $D = 2$ is $O(n)$, as shown in Section 2.1.

The links in directions other than D can be obtained from the $(D-1)$-dimensional decompositions produced during the sweep. By induction, the number of links in each $(D-1)$-decomposition is $O(n^{D-2})$, so the total number of links in directions other than D is $O(n^{D-1})$.

To count the number of links in direction D, observe how the cross section changes as the sweep hyperplane advances through the domain. If two adjacent cross sections differ by u obstacles, the number of links between the cross sections is $O(n^{D-2}u)$ by Lemma 2.8. The total number of obstacle changes during the sweep is $O(n)$, so the total number of links between all pairs of adjacent cross sections is $O(n^{D-1})$. □

We compute the links by an approach that roughly follows the proof of Lemma 2.9. As in the proof, we consider separately the links in direction D and in the other directions. The links in directions other than D can be obtained recursively from the solutions of the subproblems. The links in direction D are computed by performing an additional sweep between each pair of adjacent cross sections and using a recursive method similar to the proof above. We may again generate some links multiple times, but the total number of generated links is still $O(n^{D-1})$ so the duplicates can be removed in $O(n^{D-1} \log n)$ time.

Algorithm 2.5 shows how to find the overlapping pairs in a single cross section. The algorithm processes each set of events with equal D-coordinate as a single unit. For each unit we recursively search for new links between the newly added rectangles, as well as between old and new rectangles. We maintain the set of rectangles currently intersecting the sweep hyperplane to avoid adding duplicates to the result.

Lemma 2.10. *Algorithm 2.5 finds each intersecting pair between the input sets A and B exactly once.*

Proof. Proof by induction on D. If $D = 2$, we use Algorithm 2.3, which was shown to work correctly in Section 2.2.

Algorithm 2.5 Find all the overlapping pairs of two sets of D-dimensional rectangles.

Input: Sets of disjoint D-dimensional rectangles A and B.

Output: All the overlapping pairs of elements in A and B.

procedure FINDOVERLAPPING(D, A, B)
  if $D = 2$ **then**
    return overlapping pairs computed with Algorithm 2.3.
  end if
  $E \leftarrow$ List of all the rectangles in A and B perpendicular to D-axis.
  Sort E by D-coordinate.
  $G \leftarrow$ Groups of consecutive elements of E with equal D-coordinate.
  $P_A \leftarrow$ Empty set of references to A.
  $P_B \leftarrow$ Empty set of references to B.
  $R \leftarrow$ Empty list of pairs.
  for all $g \in G$ **do**
    $z \leftarrow$ the D-coordinate of g.
    $S_A \leftarrow$ Elements of A with D-range lower bound z.
    $S_B \leftarrow$ Elements of B with D-range lower bound z.
    $T_A \leftarrow$ Elements of A with D-range upper bound z.
    $T_B \leftarrow$ Elements of B with D-range upper bound z.
    $P_A \leftarrow P_A \backslash T_A$.
    $P_B \leftarrow P_B \backslash T_B$.
    $R \leftarrow R +$ `FindOverlapping`$(D-1, P_A, S_B)$.
    $R \leftarrow R +$ `FindOverlapping`$(D-1, S_A, P_B)$.
    $R \leftarrow R +$ `FindOverlapping`$(D-1, S_A, S_B)$.
    $P_A \leftarrow P_A \cup S_A$.
    $P_B \leftarrow P_B \cup S_B$.
  end for
  return R.
end procedure

For $D \geqslant 3$ the algorithm iterates over the D-coordinates where at least one of A or B changes. By induction we know that the three recursive calls inside the iteration each report the intersections of the provided inputs exactly once. On each iteration we create the sets S_A and S_B, which are the newly introduced hyperrectangles at this coordinate. Since S_A and S_B contain only new rectangles, none of the three recursive calls return any pairs that were previously added. Thus the algorithm finds each intersecting pair at most once.

At the point of making the recursive calls, the sets P_A and P_B contain all the rectangles whose D-ranges strictly contain the current D-coordinate. Two ranges (half-open intervals) can only intersect if they either share the lower bound or the lower bound of one of the ranges is strictly inside the other range. The recursive call `FindOverlapping`$(D-1, S_A, S_B)$ handles the ranges with the common lower bound, and the other two recursive calls handle the case of the lower bound being strictly inside the other range. Thus the algorithm finds all the intersecting pairs at least once. Since we have shown that each pair is found at most once, this completes the proof. □

To prove the running time of the link computation in a three-dimensional domain, we showed that Algorithm 2.3 works in time $O(n \log n + k)$ for any input of size n. For Algorithm 2.5 we cannot make such a generic statement, because for an arbitrary input there is no guarantee that the subproblems are significantly smaller than the original problem. Instead we prove a less generic bound by using the properties of the D-dimensional decomposition.

Lemma 2.11. *Let A and B be two decompositions generated by Algorithm 2.4 for two different D-dimensional inputs that have a total of n obstacles. Then Algorithm 2.5 runs in $O(n^{D-1} \log n + k)$ time for inputs A and B, where k is the number of returned pairs.*

Proof. Proof by induction on D. The base case $D = 2$ is proven by Lemma 2.5.

For $D \geqslant 3$ the decomposition is created by combining the $(D-1)$-dimensional decompositions at each cross section. Thus the sweep hyperplane of Algorithm 2.5 intersects a $(D-1)$-dimensional decomposition produced by Algorithm 2.4 at each point of the sweep. By Lemma 2.6 this implies that the index sets maintained during the sweep have size $O(n^{D-2})$. Performing the set operations on the n steps of the sweep then has total time complexity $O(n^{D-1} \log n)$.

The time taken by each of the three recursive calls on each cross section is $O(n^{D-2} \log n + u)$ by induction, where u is the number of overlapping pairs returned from the subproblem. By Lemma 2.10 each intersection is found exactly once, so the total complexity of the recursive calls during the sweep is $O(n^{D-1} \log n + k)$. □

Finally we can generalize Theorem 1 to higher dimensional domains.

Theorem 2. *A D-dimensional rectilinear domain defined by n obstacle faces can be decomposed into $O(n^{D-1})$ hyperrectangles with $O(n^{D-1})$ links between them in $O(n^{D-1} \log n)$ time and $O(n^{D-1})$ space.*

Proof. The hyperrectangles can be constructed in $O(n^{D-1} \log n)$ time according to Lemma 2.7. The total number of links between the cells is $k = O(n^{D-1})$ according to Lemma 2.9. Computing the links between all the layers can then be done in $O(n^{D-1} \log n + k) = O(n^{D-1} \log n)$ time by Lemma 2.11. Thus the total time complexity is $O(n^{D-1} \log n)$, and the total number of hyperrectangles and links is $O(n^{D-1})$.

Algorithm 2.4 and Algorithm 2.5 both require $O(n^{D-2})$ additional space for binary search trees and event sets, so the space complexity equals the size of the output $O(n^{D-1})$. □

3 Segment tree

A segment tree is a classical data structure that allows storing and querying intervals [5, 21]. The structure allows performing a wide range of operations efficiently. For example we can use a segment tree to find all the intervals containing a given point in time $O(\log n)$ from a collection of n intervals. We first present the general idea of the tree, and then show how it can be applied to the minimum link path problem.

Segment trees are typically implemented as semi-dynamic data structures. This means that the tree can be efficiently modified after it has been build, but we need to specify the set of possible endpoints P of intervals in advance. This allows implementing most operations in $O(\log n)$ time, and the size of the tree is $O(n)$, where $n = |P|$. The segment tree can also be generalized to create a multidimensional data structure. A D-dimensional segment tree stores D-dimensional hyperrectangles, and many operations can be supported in $O(\log^D n)$ time.

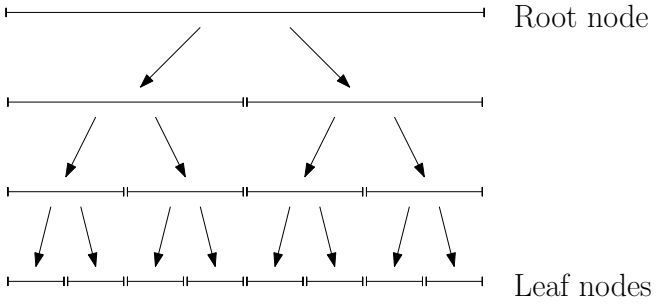


Figure 6: A segment tree is a complete or an almost complete binary tree. Each tree node is associated with an interval. The root node represents the total range, and the children of each branch node divide the range of the node into two parts.

3.1 Segment tree structure

A segment tree is a binary tree where each node s corresponds to a fixed interval $\textsc{intr}(s)$ [5]. The child nodes of each branch node divide the interval into smaller parts. There are several variations of how to exactly define the intervals of the nodes [5, 11]. In this thesis we define $\textsc{intr}(s)$ to be a half-open real number interval for each s.

Each branch node s has two child nodes, $\textsc{left}(s)$ and $\textsc{right}(s)$. The intervals of the children divide the parent interval into two parts:

$$\textsc{intr}(s) = \textsc{intr}(\textsc{left}(s)) \cup \textsc{intr}(\textsc{right}(s))$$
$$\textsc{intr}(\textsc{left}(s)) \cap \textsc{intr}(\textsc{right}(s)) = \varnothing$$

The root node corresponds to the largest supported interval $[P[1], P[n])$, and the leaf nodes correspond to the smallest possible intervals $[P[i], P[i+1])$. Figure 6 illustrates the segment tree structure.

The structure of a semi-dynamic segment tree is independent of the intervals stored in the tree. The nodes are commonly arranged into an almost complete binary tree, which allows storing the tree nodes in an array similarly to binary heap: the root node is stored in index 1, and the children of node i are $2i$ and $2i+1$. This representation allows implementing segment tree operations with very low overhead, making the segment tree a powerful practical tool in addition to providing asymptotic bounds.

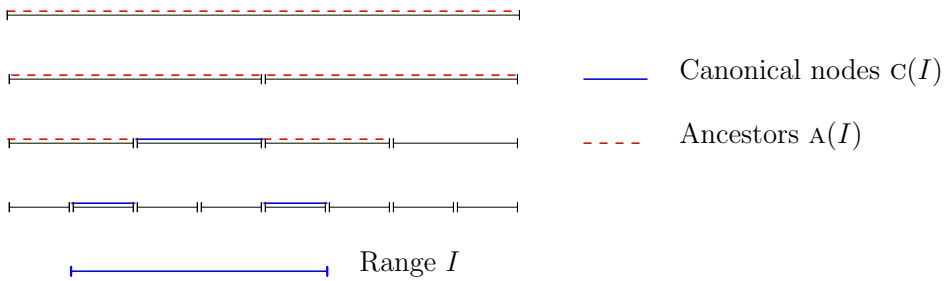


Figure 7: The canonical nodes of an interval I are the smallest set of nodes covering exactly I. The number of canonical nodes $\textsc{c}(I)$ and their ancestors $\textsc{a}(I)$ is $O(\log n)$.

3.2 Canonical nodes

When an interval I is added to a segment tree, it is stored into several tree nodes called the *canonical nodes* of I, denoted by $\textsc{c}(I)$. The canonical nodes of I are the smallest set of nodes whose intervals cover I but nothing else: $\bigcup_{c \in \textsc{c}(I)} \textsc{intr}(c) = I$. Figure 7 shows an example of canonical nodes.

The set of ancestors of $\textsc{c}(I)$ is denoted $\textsc{a}(I)$. The following two lemmas are useful for proving the running times of segment tree operations.

Lemma 3.1. $|\textsc{c}(I)| = O(\log n)$ *for any interval* I.

Proof. As the segment tree is a balanced binary tree, it has depth $\Theta(\log n)$. We show that $\textsc{c}(I)$ contains at most two nodes on each depth, which proves the claim.

If a set S contains three nodes on the same depth, we can replace the middle one with its parent node without affecting the interval covered by S. Thus S is not a minimal cover of I, and can not be $\textsc{c}(I)$. □

Lemma 3.2. $|\textsc{a}(I)| = O(\log n)$ *for any interval* I.

Proof. Similarly to the previous proof we show that $\textsc{a}(I)$ cannot contain more than two nodes on the same depth.

Suppose, for a contradiction, that $\textsc{a}(I)$ contains three nodes on the same depth: a, b and c in that order. Then $\textsc{intr}(a) \cap I \neq \varnothing$ and $\textsc{intr}(c) \cap I \neq \varnothing$, so $\textsc{intr}(b) \subseteq I$. Some descendant of b belongs to $\textsc{c}(I)$, so $\textsc{c}(I)$ is not a minimal cover, which is a contradiction. □

We can iterate over the canonical nodes and their ancestors by using Algorithm 3.1 starting from the root node.

Algorithm 3.1 Visit all the canonical nodes of an interval and their ancestors.

Input: Interval I.

Effect: Calls `VisitCanonical` and `VisitAncestor` for all the elements in $\textsc{c}(I)$ and in $\textsc{a}(I)$ respectively.

```
procedure ITERCANONICAL(s)          ▷ Visit nodes in the subtree rooted at s
    if INTR(s) ⊆ I then
        VisitCanonical(s).
    else if INTR(s) ∩ I ≠ ∅ then
        VisitAncestor(s).
        IterCanonical(LEFT(s)).
        IterCanonical(RIGHT(s)).
    end if
end procedure
```

Algorithm 3.1 can be used to implement operations such as adding an interval to a segment tree by defining the methods `VisitCanonical` and `VisitAncestor` appropriately. The recursion traverses through the nodes in $\textsc{a}(I)$, and stops immediately when it arrives either to a node in $\textsc{c}(I)$ or to a node neither in $\textsc{c}(I)$ nor in $\textsc{a}(I)$. Thus the running time is proportional to the sizes of $\textsc{c}(I)$ and $\textsc{a}(I)$, which is $O(\log n)$ by Lemma 3.1 and Lemma 3.2.

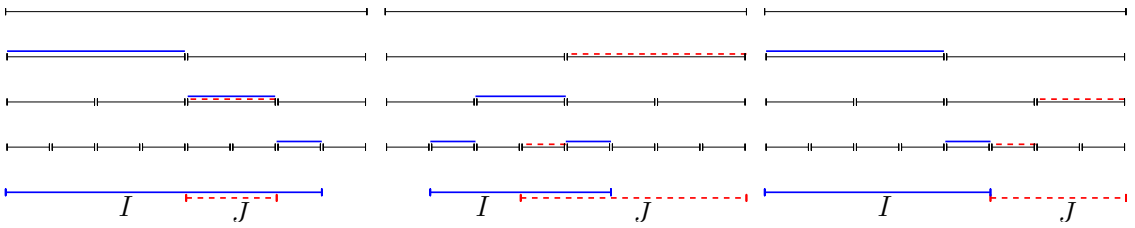
3.3 Tree operations

We now show how the segment tree can be used for detecting intersecting intervals [5, Chapter 10]. The following lemma gives the necessary and sufficient conditions for detecting intersecting intervals by using the canonical nodes. Figure 8 illustrates the different cases of the lemma.

Lemma 3.3. *Consider any two intervals I and J. At least one of the following is true if and only if $I \cap J \neq \varnothing$.*

1. $\textsc{c}(I) \cap \textsc{c}(J) \neq \varnothing$,
2. $\textsc{c}(I) \cap \textsc{a}(J) \neq \varnothing$,
3. $\textsc{a}(I) \cap \textsc{c}(J) \neq \varnothing$.

Proof. First suppose that $I \cap J \neq \varnothing$. Let x be any leaf node of the segment tree such that $\textsc{intr}(x) \subseteq I \cap J$. Let A be the set of ancestors of x. Since the canonical



(a) $\text{C}(I) \cap \text{C}(J) \neq \varnothing$ (b) $\text{C}(I) \cap \text{A}(J) \neq \varnothing$ (c) $I \cap J = \varnothing$

Figure 8: Different cases of Lemma 3.3. Two intervals I and J and their canonical nodes are drawn in each subfigure. If a pair of intervals intersect, they either have a common canonical node (Fig. 8a) or an ancestor of a canonical node of one interval is a canonical node of the other interval (Fig. 8b). If the intervals don't intersect, the canonical sets are also separate (Fig. 8c).

nodes of an interval cover the whole interval, A intersects both $\text{C}(I)$ and $\text{C}(J)$. Let i and j be the nodes where A intersects $\text{C}(I)$ and $\text{C}(J)$ respectively. Either i and j are the same node, or one of them is an ancestor of the other, which proves the "only if" part of the claim.

For the "if" part, we look at the three cases separately.

1. If there exists a node $s \in \text{C}(I) \cap \text{C}(J)$, then $\text{INTR}(s) \subseteq I \cap J$, so $I \cap J \neq \varnothing$.
2. If there is a node $s \in \text{C}(I) \cap \text{A}(J)$, then there exists $m \in \text{C}(J)$ that is a descendant of s. $\text{INTR}(m) \subseteq J$ and $\text{INTR}(m) \subsetneq \text{INTR}(s) \subseteq I$, so $I \cap J \neq \varnothing$.
3. Identical to the second case.

□

Lemma 3.3 can be used to design an algorithm for detecting intersecting pairs of intervals in a segment tree. We maintain two lists of intervals in each segment tree node s:

- $\text{IC}(s)$: Intervals I for which $s \in \text{C}(I)$,
- $\text{IP}(s)$: Intervals I for which $s \in \text{A}(I)$.

When an interval I is added to the segment tree, it is added to $\text{IC}(s)$ for all $s \in \text{C}(I)$ and to $\text{IP}(s)$ for all $s \in \text{A}(I)$. When we look for intervals intersecting a given interval J, we report all the intervals in $\text{IC}(s)$ for all $s \in \text{A}(J)$ and in $\text{IP}(s)$ for all $s \in \text{C}(J)$. By

Lemma 3.3 this finds exactly the intervals intersecting J and nothing else. Note that some intervals might be found multiple times as they are split into $O(\log n)$ canonical nodes in the tree. The structure can be further extended to avoid reporting the duplicates, but for the minimum link paths use case we only need to check whether the intersecting set is empty or not, so this solution is sufficient.

We also want to support clearing an interval from the tree. Clearing an interval I means that each interval J in the tree is replaced by $J \backslash I$, potentially removing J or cutting it into two parts. The clear operation is performed in the following three steps:

1. Push down the intervals stored in each $s \in \text{A}(I)$ into $\text{LEFT}(s)$ and $\text{RIGHT}(s)$.
2. Clear each subtree rooted in any node $s \in \text{C}(I)$.
3. Recompute $\text{IP}(s)$ to match the remaining intervals for all nodes $s \in \text{A}(I)$.

All the tree operations can be done in parallel during a single recursive traversal of the tree. For nodes $s \in \text{A}(I)$ we push down the intervals in $\text{IC}(s)$ to the child nodes before proceeding with the recursion. When we arrive in a node $s \in \text{C}(I)$, the entire subtree rooted at s is cleared. The recursion can be stopped early if the current subtree is empty already. After the child nodes are cleared, we also need to regenerate the $\text{IP}(s)$ list for the nodes $s \in \text{A}(I)$ to remove the intervals that are no longer present in the subtree. The regeneration is done by merging the lists in the child nodes of s. The exact process is shown in Algorithm 3.2.

The running time of Algorithm 3.2 depends on the time spent in copying and merging the intervals stored in the tree nodes. If we are only interested in querying whether a given interval overlaps any interval in the tree rather than finding the overlapping intervals, we can replace the interval lists by booleans indicating whether the list is non-empty. We analyze this more simple case, as it is sufficient for the minimum link path algorithms.

Lemma 3.4. *Suppose that the copying and merging of interval sets in Algorithm 3.2 can be done in constant time. Then the time complexity of clearing range I is $O(\log n + k)$, where k is the number of cleared nodes.*

Proof. The function `ClearInterval` traverses through nodes $\text{C}(I)$, $\text{A}(I)$ and the descendants of the canonical nodes, stopping immediately if it ends up in an empty subtree. In each descendant node we either clear the node or return immediately, so

Algorithm 3.2 Clear interval I from a segment tree.

Input: Interval I.

Effect: Clears I from the segment tree.

```
procedure ClearInterval(s)
    if INTR(s) ∩ I = ∅ or (IP(s) and IC(s) are empty) then
        return
    end if
    if INTR(s) ⊈ I then                                   ▷ s ∈ A(I)
        IC(LEFT(s)) ← IC(LEFT(s)) ∪ IC(s).
        IC(RIGHT(s)) ← IC(RIGHT(s)) ∪ IC(s).
    end if
    Clear IC(s).
    Clear IP(s).
    if LEFT(s) and RIGHT(s) are defined then
        ClearInterval(LEFT(s)).
        ClearInterval(RIGHT(s)).
    end if
    if INTR(s) ⊈ I then                                   ▷ s ∈ A(I)
        IP(s) ← IP(LEFT(s)) ∪ IP(RIGHT(s)) ∪ IC(LEFT(s)) ∪ IC(RIGHT(s)).
    end if
end procedure
```

we only visit as many descendants as there are cleared nodes. The number of canonical nodes and their ancestors is $O(\log n)$, so the time complexity is $O(\log n + k)$. □

3.4 Multidimensional segment tree

The segment tree can be generalized into a two or higher dimensional structure [18]. A two-dimensional segment tree allows storing rectangles and performing efficient queries for rectangular regions. Correspondingly a D-dimensional segment tree allows performing efficient operations for D-dimensional hyperrectangles. We use the terms rectangle and hyperrectangle interchangeably to refer to a hyperrectangle of any dimension D.

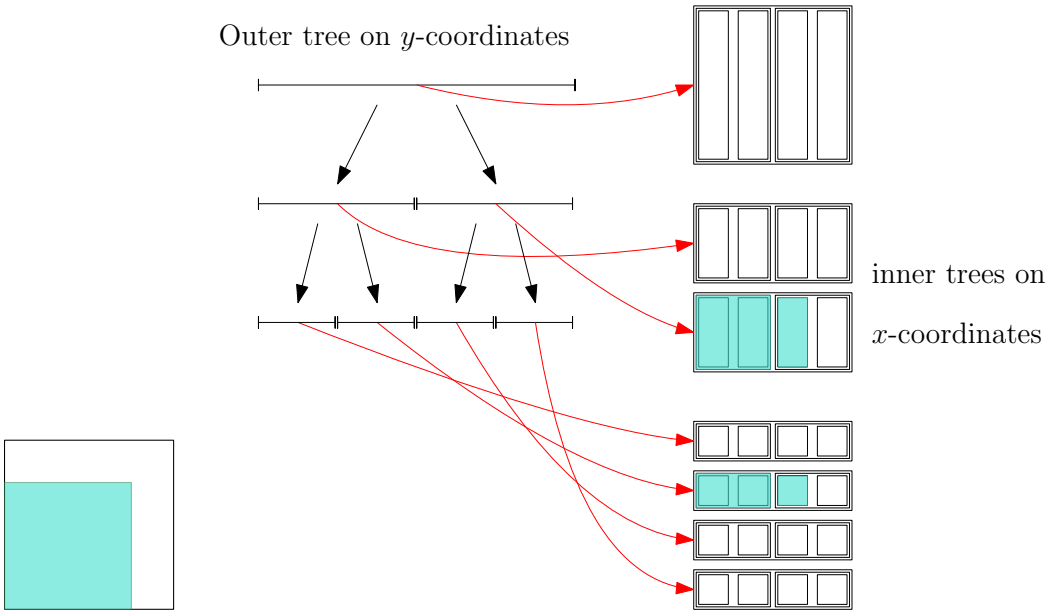
A multidimensional segment tree is composed of nested regular segment trees. The shape of a D-dimensional segment tree is a one-dimensional segment tree whose each node stores a $(D-1)$-dimensional segment tree.

Each tree dimension represents one of the D coordinate axes. Each node s of the innermost segment tree represents a D-dimensional hyperrectangle $\text{RECT}(s)$, whose bounds are defined by the position of the node in each of the D tree layers. For example we can define the outer tree of a two-dimensional segment tree to represent the y coordinates, and each inner tree to represent the x coordinates. Then each node of the inner tree represents a rectangle whose x-range is determined by the position of the node in the inner tree, and the y-range is determined by the node of the outer tree that contains the inner tree.

The concept of canonical nodes can also be extended to higher dimensions. The canonical nodes $\text{C}(R)$ of a D-dimensional rectangle R are the smallest set of nodes that fully cover R but nothing else. Figure 9 illustrates the structure of a two-dimensional segment tree and the canonical nodes.

In order to compute the canonical nodes we look into each axis separately. Let interval R_i be the projection of rectangle R into i-axis. Let $\text{C}_i(R_i)$ be the canonical nodes of R_i in the ordinary segment tree built for i-coordinates.

Consider a two-dimensional segment tree and two arbitrary nodes of the projected trees $c_1 \in \text{C}_1(R_1)$ and $c_2 \in \text{C}_2(R_2)$. Let $c_1 \times c_2$ be the node of the two-dimensional tree that is found in position of c_1 in the inner tree, and c_2 in the outer tree. Similarly for a D-dimensional tree we define the product $c_1 \times c_2 \times \cdots \times c_D$ to be the tree node indicated by positions of the one-dimensional tree nodes. The following lemma shows how the canonical nodes of the projections can be combined to find the canonical



(a) Rectangle R relative to total area

(b) The canonical nodes of R in a two-dimensional segment tree

Figure 9: A rectangle is split to canonical nodes in a two-dimensional segment tree. Each node of the outer tree defines a range of y-coordinates and an inner segment tree built on the x-coordinates. Each node of the inner trees is a rectangle that combines the y and the x ranges. The canonical nodes of a rectangle R (Fig. 9a) are the smallest set of nodes of the inner trees that cover exactly R (Fig. 9b).

nodes in a D-dimensional tree.

Lemma 3.5. *For any D-dimensional rectangle R*

$$\textsc{c}(R) = \{c_1 \times c_2 \cdots \times c_D \mid c_1 \in \textsc{c}_1(R_1), \ldots, c_D \in \textsc{c}_D(R_D)\}.$$

In other words, $\textsc{c}(R)$ is the Cartesian product of one-dimensional canonical sets

$$\textsc{c}(R) = \bigtimes_{i=1}^{D} \textsc{c}_i(R_i).$$

Proof. Proof by induction on D. The base case $D = 1$ follows directly from the definition of $\textsc{c}(R)$.

For $D \geqslant 2$ note that all of the inner $(D-1)$-dimensional segment trees are identical. Thus to cover a rectangular region minimally, we should select the same set of nodes in each inner tree. By induction we should select the nodes $\bigtimes_{i=1}^{D-1} \textsc{c}_i(R_i)$ in the inner trees to cover axes $1 \ldots D-1$ optimally. To cover also axis D, it is optimal to select the nodes $\textsc{c}_D(R_D)$, and thus $\textsc{c}(R) = \bigtimes_{i=1}^{D} \textsc{c}_i(R_i)$. □

Combination of Lemma 3.5 and Lemma 3.1 gives a bound for the number of canonical nodes.

Corollary 3.6. *$|\textsc{c}(R)| = O(\log^D n)$ for any D-dimensional rectangle R, where n is an upper bound for the number of supported endpoints in any of the dimensions.*

Proof.

$$|\textsc{c}(R)| = \left| \bigtimes_{i=1}^{D} \textsc{c}_i(R_i) \right| = \prod_{i=1}^{D} |\textsc{c}_i(R_i)| = \prod_{i=1}^{D} O(\log n) = O(\log^D n).$$

□

For regular segment trees there is a clear connection between the parent-child hierarchy and interval overlap. For any two tree nodes a and b, $\textsc{intr}(a) \subseteq \textsc{intr}(b)$ holds if and only if b is an ancestor of a in the tree. In a multidimensional case it also applies that if b is an ancestor of a, then $\textsc{rect}(a) \subseteq \textsc{rect}(b)$. However the reverse is not true; $\textsc{rect}(a) \subseteq \textsc{rect}(b)$ does not imply that b is an ancestor of a because b can be in a different subtree in some of the outer trees.

Instead of looking for parent-child relationships directly in the tree, we can look at them separately on each coordinate axis. This allows us to generalize Lemma 3.3 for segment interaction to work for D-dimensional rectangles.

Lemma 3.7. *Consider any two D-dimensional rectangles A and B. If $A \cap B \neq \varnothing$, then at least one of the following holds for each coordinate axis $d \in \{1, \ldots, D\}$.*

1. $\mathrm{C}_d(A_d) \cap \mathrm{C}_d(B_d) \neq \varnothing$,
2. $\mathrm{C}_d(A_d) \cap \mathrm{A}_d(B_d) \neq \varnothing$,
3. $\mathrm{A}_d(A_d) \cap \mathrm{C}_d(B_d) \neq \varnothing$.

Furthermore, if $A \cap B = \varnothing$, then there is at least one coordinate axis for which none of the above conditions hold.

Proof. A and B overlap if and only if their projections to each coordinate axis overlap. Suppose that $A \cap B \neq \varnothing$. Then A_d and B_d overlap for each d, and by Lemma 3.3 the condition holds for each d. If $A \cap B = \varnothing$, then there exists some d for which $A_d \cap B_d = \varnothing$, so again by Lemma 3.3 none of the conditions hold for d. □

To find pairs of rectangles fulfilling the conditions of Lemma 3.7, we store *two* $D-1$ dimensional inner trees in every outer node of a D-dimensional segment tree:

`subtree`(s) where we store rectangles R for which $s \in \mathrm{C}_D(R_D)$,

`subtreeP`(s) where we store rectangles R for which $s \in \mathrm{A}_D(R_D)$.

We can implement rectangle insertion and query operations by using these fields. Algorithm 3.3 shows the insert operation and Algorithm 3.4 shows the query operation. For simplicity we only support checking whether a query rectangle intersects any rectangle in the tree rather than returning the matching rectangles. The nodes s of the innermost tree have dimension 0, and they contain only a single boolean `hasrect`(s), which indicates that at least one rectangle is inserted to this subtree.

Lemma 3.8. *`InsertToTree` and `QueryTree` both have time complexity $O(\log^D n)$.*

Proof. Proof by induction on D. For the base case $D = 0$ both algorithms finish in constant time. For $D \geqslant 1$, the algorithms iterate over the canonical nodes and their ancestors in the outer tree, making $\log n$ calls to the inner trees. By induction the inner tree operations are done in $O(\log^{D-1} n)$ time, so the time complexity is $O(\log^D n)$. □

Algorithm 3.3 Add a rectangle to a D-dimensional segment tree.

Input: D-dimensional rectangle R.

Effect: Sets $\texttt{hasrect}(s)$ for each canonical node s of R.

```
procedure InsertToTree(D, s)          ▷ Insert to the subtree rooted at node s
    if D = 0 then
        hasrect(s) ← true.
    else if INTR(s) ⊆ R_D then
        InsertToTree(D − 1, subtree(s)).
    else if INTR(s) ∩ R_D ≠ ∅ then
        InsertToTree(D − 1, subtreeP(s)).
        InsertToTree(D, LEFT(s)).
        InsertToTree(D, RIGHT(s)).
    end if
end procedure
```

Algorithm 3.4 Query whether a rectangle intersects with any rectangle in a multi-dimensional segment tree.

Input: D-dimensional rectangle R.

Output: Whether $\texttt{hasrect}(s)$ is true for any node s matching R.

```
procedure QueryTree(D, s)             ▷ Check subtree rooted at node s
    if D = 0 then
        return hasrect(s).
    else if INTR(s) ⊆ R_D then
        return QueryTree(D − 1, subtree(s))
            or QueryTree(D − 1, subtreeP(s)).
    else if INTR(s) ∩ R_D ≠ ∅ then
        return QueryTree(D − 1, subtree(s))
            or QueryTree(D, LEFT(s))
            or QueryTree(D, RIGHT(s)).
    end if
end procedure
```

Note that `subtree`() and `subtreeP`() are used in opposite ways in the two operations. In `InsertToTree` we recurse down to `subtree`() for canonical nodes, and to `subtreeP`() for their ancestors, whereas in `QueryTree` we recurse to `subtree`() in the ancestor nodes, and to both of the trees in the canonical nodes. This allows us to ensure that `InsertToTree` and `QueryTree` arrive in the same node if and only if the conditions of Lemma 3.7 are fulfilled.

Suppose we first add a rectangle A and then query for a rectangle B in the tree. The `QueryTree` operation returns true if the two operations end up in any common dimension 0 node. The only way for the two operations to end up in a common node is if in each dimension d we are in a canonical node of at least one of A_d and B_d, and in either a canonical node or in an ancestor of a canonical node for the other rectangle.

Thus we can efficiently support inserting and querying for D-dimensional rectangles by a nested structure consisting of multiple levels of segment trees. However clearing a rectangle in this structure is a more complex operation. To make the clear operation simpler, we turn into another variant of the multidimensional segment tree, the unified segment tree.

3.5 Unified segment tree

The structure of a multidimensional segment tree depends on the order of the coordinate axes. For example a two-dimensional segment tree can either have the outer tree represent y-coordinates and each inner tree represent x-coordinates, or the other way round. The structure and the parent-child relationships are different in each case, but the set of rectangles representable in the tree is exactly the same regardless of the order.

The *unified segment tree* (also just *unified tree*), developed by Wagner [20], combines the parent-child links of the different multidimensional segment trees into a single structure. The unified segment tree is actually not a tree but a directed graph, but we use the word tree because of the close resemblance to the segment tree. Each node s of the graph represents a D-dimensional hyperrectangle. For each axis i where $\textsc{rect}(s)$ is not minimal, s has links to two child nodes that divide $\textsc{rect}(s)$ into two parts along a hyperplane perpendicular to the i-axis. Thus each node has between 0 and $2D$ links to child nodes. The children splitting the node s along axis i are denoted $\textsc{fst}_i(s)$ and $\textsc{snd}_i(s)$. Figure 10 illustrates the structure of the unified

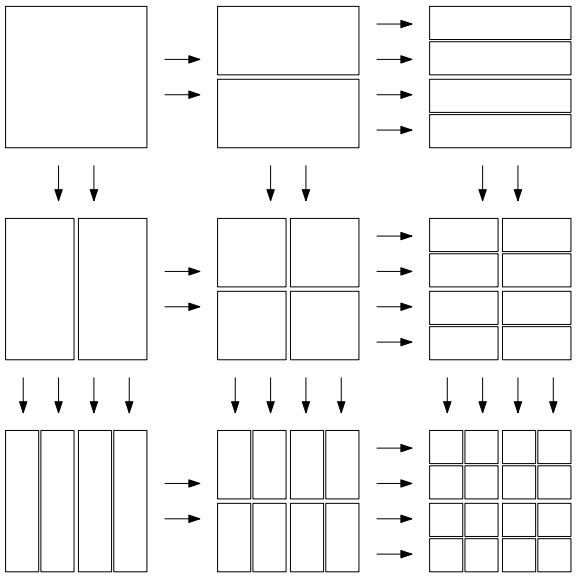


Figure 10: The nodes of a unified segment tree form a directed acyclic graph. Each node can be split along any of the coordinate axes as long as the node is not minimal in that direction.

segment tree in two-dimensional case.

For our purposes, the main advantage of the unified segment tree over a regular multidimensional segment tree is that clearing a rectangle from the tree is more straightforward. The basic idea of clearing a hyperrectangle is similar to Algorithm 3.2 for clearing an interval. In the multidimensional case we need to split partially cleared large rectangles into child rectangles along multiple axes. The segment tree structure is not well suited for this task, but in the unified segment tree the splitting is easy.

Note that in the unified tree there is no distinction between outer and inner tree. In the multidimensional segment tree the nodes of a nested tree represent ranges in one of the coordinate axes, and the innermost nodes represent the Cartesian products of the ranges. In the unified tree there is no concept of outer and inner nodes, and each node represents a D-dimensional rectangle.

The operations `InsertToTree` and `QueryTree` for the unified tree are similar to the algorithms for the regular multidimensional tree. We consider again querying whether a rectangle intersects with any of the rectangles in the tree. Each tree node stores 2^D bits of information about the rectangles stored in the subtree. The bit array stored in node s is denoted `hasbits`(s). Initially none of the bits are set. When a rectangle R is added to a node s, the bits are used to define whether s represents a canonical node or an ancestor of a canonical node along each coordinate axis. The

2^D bits allow us to detect when the conditions of Lemma 3.7 become fulfilled when querying for intersecting rectangles.

Algorithm 3.5 defines function `NodeMask`() that computes a bitmask of D bits for a rectangle and a tree node. This mask is used to determine which bits of `hasbits`(s) to access when adding a rectangle or querying the tree.

Algorithm 3.5 Compute the bitmask representing whether a node is covered by a rectangle in each direction.

Input: D-dimensional rectangle R and segment tree node s.
Output: Bitmask of D bits representing the dimensions where R fully covers s.

```
procedure NODEMASK(R, s)
    m ← 0.
    for all d ∈ 1 ... D do
        if RECT(s)_d ⊆ R_d then
            m ← m | 2^(d−1).
        end if
    end for
    return m.
end procedure
```

We can use the `NodeMask` function to implement the insert and query operations for the unified segment tree. Algorithm 3.6 shows the tree insert operation.

Algorithm 3.7 shows the implementation of the query operation. The function `invertBits` reverses the bits of a D-bit input. For example if $D = 4$ then `invertBits`$(0111_2) = 1000_2$.

Lemma 3.9. *Algorithm 3.6 and Algorithm 3.7 both have time complexity* $O(\log^D n)$.

Proof. The proof is similar to Lemma 3.8. The algorithms use recursion with argument d defining the axis along which the tree is traversed. Initially the recursive functions are called with argument $d = D$. We prove the claim by induction on d.

For the base case $d = 0$ both algorithms finish in constant time. For $d \geqslant 1$, the algorithms iterate over the canonical nodes and their ancestors by moving along axis d, making $O(\log n)$ calls with argument $d - 1$. By induction the operations with argument $d-1$ are done in $O(\log^{d-1} n)$ time, so the time complexity is $O(\log^d n)$. □

Both `InsertToTree` and `QueryTree` iterate over the canonical nodes of the input rectangle as well as their ancestors, and access `hasbits`(s) for each such node

Algorithm 3.6 Add a rectangle to a D-dimensional unified tree.

Input: D-dimensional hyperrectangle R.

Effect: Sets `hasbits`$(s)[b]$ for each node s and bitmask b matching R.

```
procedure INSERTTOTREE(d, s)        ▷ Insert to the subtree rooted at node s by moving along axes 1 ... d
    if d = 0 then
        m ← NodeMask(R, s).
        for all b ∈ 0 ... 2^D − 1 do
            if All bits of b are set in m then
                hasbits(s)[b] ← true.
            end if
        end for
    else if RECT(s)_d ⊆ R_d then
        InsertToTree(d − 1, s).
    else if RECT(s)_d ∩ R_d ≠ ∅ then
        InsertToTree(d − 1, s).
        InsertToTree(d, FST_d(s)).
        InsertToTree(d, SND_d(s)).
    end if
end procedure
```

Algorithm 3.7 Query whether a rectangle intersects with any rectangle in the unified tree.

Input: D-dimensional hyperrectangle R.

Output: Whether `hasbits`$(s)[b]$ is set for any node s and bitmask b matching R.

```
procedure QUERYTREE(d, s)                ▷ Check subtree rooted at node s
    if d = 0 then
        return hasbits(s)[invertBits(NodeMask(R, s))].
    else if RECT(s)_d ⊆ R_d then
        return QueryTree(d − 1, s).
    else if RECT(s)_d ∩ R_d ≠ ∅ then
        return QueryTree(d − 1, s)
            or QueryTree(d, FST_d(s))
            or QueryTree(d, SND_d(s)).
    end if
end procedure
```

s. When adding a rectangle A, we set $\texttt{hasbits}(s)[b]$ for each bitmask b whose bits are covered by $\texttt{NodeMask}(A, s)$. To query for a rectangle B, we check the bit $\texttt{hasbits}(s)[\texttt{invertBits}(\texttt{NodeMask}(B, s))]$ for each accessed node s. This expression detects A if $\texttt{NodeMask}(A, s)|\texttt{NodeMask}(B, s) = 2^D - 1$, where $x|y$ represents the bitwise OR operation of x and y. This is equivalent to the condition that $\textsc{rect}(s)_d$ is the range of a canonical node of either A_d or B_d or both in each dimension d. This matches the conditions of Lemma 3.7 for rectangle intersection, so the match is returned if and only if A and B intersect.

Next we consider clearing a rectangle from the unified tree. Clearing a rectangle R means that all the rectangles in the tree are cut such that the region of R becomes empty. The clear operation is implemented using the same three operations as we used for the segment tree clearing in Section 3.3: push down any intervals in the ancestors, clear the subtrees of the canonical nodes, and finally recompute the information about rectangles stored in the descendants.

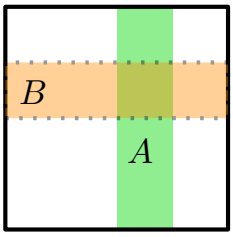
The three operations are performed by a recursive function `ClearRectangle` that traverses through the tree. The function uses two arguments to control the tree traversal: the current subtree root s, and the axis where we are moving d. In each canonical ancestor $s \in \textsc{a}_d(R_d)$ we split the contents of the subtree along axis d before proceeding, and recompute the descendant information after finishing clearing the subtree. Figure 11 shows an example of the clearing process.

Algorithm 3.8 shows the pseudocode of the clear algorithm. The values are pushed down in the tree by calling a helper function `PushDownSubtree` defined in Algorithm 3.9. In the end we regenerate the bits describing the values in the descendants by using another helper function `ComputeSubtreeData` defined in Algorithm 3.10.

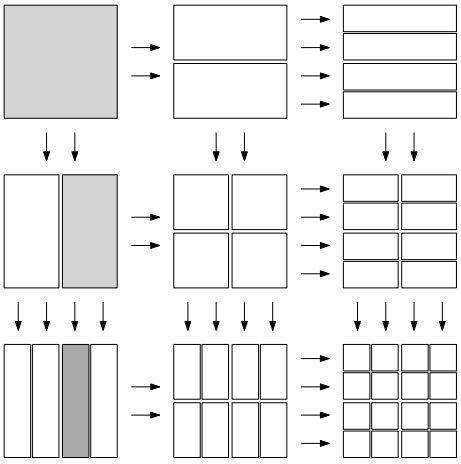
Lemma 3.10. *Algorithm 3.8 has running time $O(n^{D-1} \log n + k)$, where k is the number of cleared nodes.*

Proof. We start by proving that `PushDownSubtree` and `ComputeSubtreeData` have running time $O(n^d)$ by induction on the argument d. Clearly both functions perform $O(1)$ work when $d = 0$, which proves the base case. If $d \geqslant 1$, both functions recursively iterate over the descendants in direction d, making $O(n)$ recursive calls with argument $d - 1$. By induction the recursive calls are done in $O(n^{d-1})$ time, so the time complexity of both `PushDownSubtree` and `ComputeSubtreeData` is $O(n^d)$.

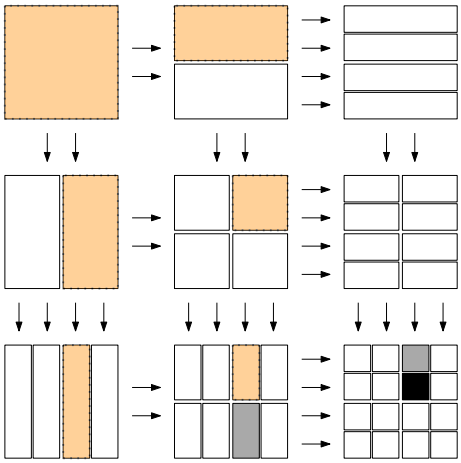
Next we prove the running time of `ClearRectangle` by induction on argument d. When called with argument d, the function `ClearRectangle` calls functions



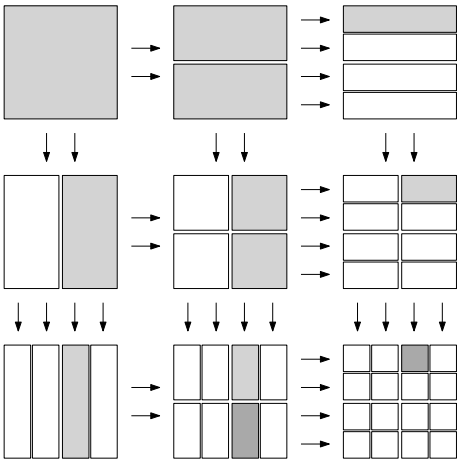
(a) We first insert A and then clear B.



(b) The only canonical node of A in dark grey and its ancestors in light grey in the unified segment tree.



(c) `PushDownSubtree` is performed in all the orange rectangles, shifting A to dark grey rectangles. The black rectangle is cleared.



(d) `ComputeSubtreeData` visits all the ancestors of the remaining nodes, marking information about the descendants into the light-grey nodes.

Figure 11: Example of clearing a rectangle in a unified segment tree using Algorithm 3.8. The rectangle A (Fig. 11a) is stored in a single canonical node and two ancestors (Fig. 11b). To clear the rectangle B, we first push A down into descendants from nodes touched by B (Fig. 11c). After clearing we fill descendant information into ancestors of the remaining parts of A (Fig. 11d).

Algorithm 3.8 Clear a rectangle from a unified segment tree.

Input: D-dimensional hyperrectangle R.

Effect: Modifies `hasbits`() arrays so that the region of R becomes empty.

```
procedure CLEARRECTANGLE(d, s)
    if RECT(s)_d ∩ R_d = ∅ or ¬hasbits(s)[0] then
        return
    end if
    if d = 0 then
        hasbits(s)[0 ... 2^(D−1)] ← false.
        return
    end if
    if RECT(s)_d ⊈ R_d then                          ▷ s_d ∈ A(R_d)
        PushDownSubtree(d − 1, s, d).
    end if
    ClearRectangle(d − 1, s).
    if FST_d(s) and SND_d(s) are defined then
        ClearRectangle(d, FST_d(s)).
        ClearRectangle(d, SND_d(s)).
    end if
    if RECT(s)_d ⊈ R_d then                          ▷ s_d ∈ A(R_d)
        ComputeSubtreeData(d, s).
    end if
end procedure
```

Algorithm 3.9 Helper procedure for `ClearRectangle`: Push down values in a subtree.

Input: Rectangle R (global), unified tree node s, current axis d and split axis a.

Effect: Copies $\texttt{hasbits}(t)$ to $\texttt{hasbits}(\text{FST}_a(t))$ and to $\texttt{hasbits}(\text{SND}_a(t))$ for all nodes t that can be accessed from s by moving along axes $1 \ldots d$.

procedure PUSHDOWNSUBTREE(d, s, a)
  if $d = 0$ **then**
    $\texttt{hasbits}(\text{FST}_a(s)) \leftarrow \texttt{hasbits}(\text{FST}_a(s)) | \texttt{hasbits}(s)$.
    $\texttt{hasbits}(\text{SND}_a(s)) \leftarrow \texttt{hasbits}(\text{SND}_a(s)) | \texttt{hasbits}(s)$.
    return
  else if $\text{RECT}(s)_d \cap R_d = \varnothing$ **then**
    return
  end if
  `PushDownSubtree`$(d - 1, s, a)$.
  if $\text{FST}_d(s)$ and $\text{SND}_d(s)$ are defined **then**
    `PushDownSubtree`$(d, \text{FST}_d(s), a)$.
    `PushDownSubtree`$(d, \text{SND}_d(s), a)$.
  end if
end procedure

Algorithm 3.10 Helper procedure for `ClearRectangle`: Recompute data about the rectangles stored in the descendant nodes.

Input: Rectangle R (global), unified tree node s and current axis d.

Effect: Generates $\texttt{hasbits}(t)$ based on the descendants of t for all nodes t that can be accessed from s by moving along axes $1 \dots d$.

```
procedure ComputeSubtreeData(d, s)
    if d = 0 then
        ComputeSubtreeDataForNode(s).
        return
    else if RECT(s)_d ∩ R_d = ∅ then
        return
    end if
    ComputeSubtreeData(d − 1, s).
    if FST_d(s) and SND_d(s) are defined then
        ComputeSubtreeData(d, FST_d(s)).
        ComputeSubtreeData(d, SND_d(s)).
    end if
end procedure
procedure ComputeSubtreeDataForNode(s)
    hasbits(s)[0 ... 2^{D−1}] ← false.
    for all a ∈ 1 ... D do
        if RECT(s)_a ⊆ R_a then
            Continue.
        end if
        ▷ RECT(s)_a is not minimal, so FST_a(s) and SND_a(s) are defined.
        c_1 ← FST_a(s).
        c_2 ← SND_a(s).
        for all b ∈ 0 ... 2^D − 1 do
            if b does not have bit a set then
                hasbits(s)[b] ← hasbits(s)[b] | hasbits(c_1)[b] | hasbits(c_2)[b].
            end if
        end for
    end for
end procedure
```

`PushDownSubtree` and `ComputeSubtreeData` with argument $d-1$. If $d = 1$ the calls to `PushDownSubtree` and `ComputeSubtreeData` perform $O(1)$ work, and the algorithm is equivalent to Algorithm 3.2 for 1-dimensional segment tree removal. Thus the base case $d = 1$ is proven by Lemma 3.4.

In case $d \geqslant 2$ we iterate over the descendants in direction d. We call `PushDownSubtree` and `ComputeSubtreeData` with argument $d-1$ for every node in $\textsc{a}_d(R_d)$. We also make $O(n)$ calls to `ClearRectangle` with argument $d-1$. The size of $\textsc{a}_d(R_d)$ is $O(\log n)$, so by induction the time complexity is $O((\log n)n^{d-1} + n(n^{d-2}\log n) + k) = O(n^{d-1}\log n + k)$. □

4 Minimum link paths in the plane

In this section we study the rectilinear minimum link path problem in the plane. We present a simple algorithm that runs in $O(n \log n)$ time and $O(n)$ space [15]. The ideas developed for the planar case are later generalized to solve the more difficult three and higher dimensional minimum link path problems in Section 5.

4.1 Intersection graph

We solve the minimum link path problem by applying the staged illumination paradigm outlined in Section 1.2 [15]. We initialize $\textsc{reach}(0)$ to contain the starting point $\mathbf{s}$, and iteratively expand the illuminated area by computing $\textsc{reach}(k+1)$ based on $\textsc{reach}(k)$. The iteration is continued until we find the endpoint $\mathbf{t}$.

Clearly any minimum link path consists of alternation between horizontal and vertical links. We split the problem into two subproblems, based on whether the first link is horizontal or vertical. The minimum link path can be found by solving both subproblems and choosing the shorter of the two paths.

Consider the case with the first link horizontal. It is easy to see that $\textsc{reach}(k+1)$ consists of points that can be reached by starting from $\textsc{reach}(k)$ and moving either horizontally or vertically, depending on the parity of k. The following lemma shows that the regions illuminated on each step can be expressed by using the horizontal and vertical decompositions defined in Section 2.1.

Lemma 4.1. *For even $k \geqslant 2$, the region $\textsc{reach}(k+1)$ consists of the rectangles $r \in \textsc{dec}_x$ that intersect $\textsc{reach}(k)$. For odd $k \geqslant 3$ the same applies for $\textsc{dec}_y$.*

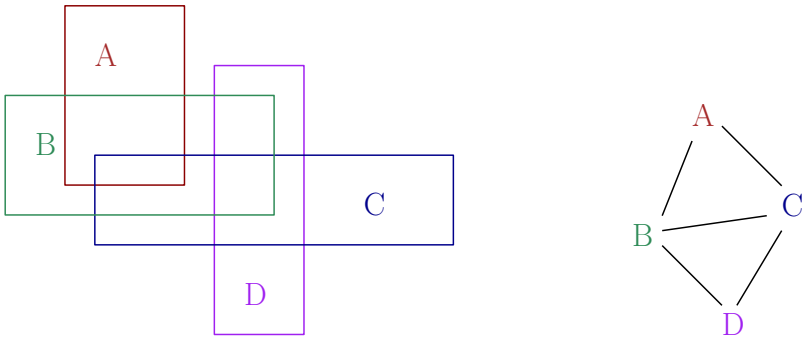


Figure 12: A group of four rectangles and their intersection graph. Each pair of overlapping objects has an edge between them in the intersection graph.

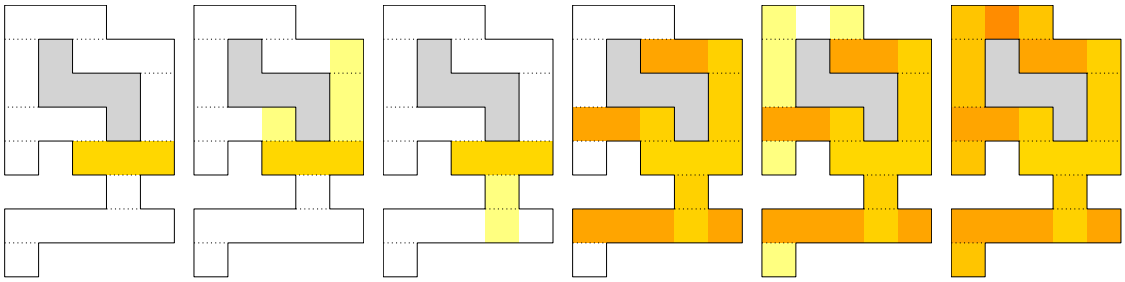
Proof. Let $k \geqslant 2$ be even. The region $\textsc{reach}(k+1)$ is formed by illuminating in horizontal direction from $\textsc{reach}(k)$.

Let h be any rectangle in $\textsc{dec}_x$ that is at least partially illuminated in $\textsc{reach}(k)$. Since $\textsc{reach}(k)$ is formed by illuminating vertically from $\textsc{reach}(k-1)$, there must be a vertical line passing through h that is contained in $\textsc{reach}(k)$. Illuminating horizontally from the vertical line illuminates h entirely, so $h \subseteq \textsc{reach}(k+1)$. Thus each rectangle of $\textsc{dec}_x$ intersected by $\textsc{reach}(k)$ is entirely contained in $\textsc{reach}(k+1)$.

For odd $k \geqslant 3$ we form $\textsc{reach}(k+1)$ by illuminating in vertical direction from $\textsc{reach}(k)$, and the same proof applies for $\textsc{dec}_y$. □

For a set of geometric objects (such as rectangles) G, the *intersection graph* of G is a graph whose nodes are the elements of G, and there is an edge between each pair of intersecting objects. Figure 12 shows an example of an intersection graph. Consider the intersection graph of $\textsc{dec}_x \cup \textsc{dec}_y$. This graph has rectangles as nodes, and intersecting rectangles between $\textsc{dec}_x$ and $\textsc{dec}_y$ are joined by an edge. Let k be an even number and $r \in \textsc{dec}_y$ contained in $\textsc{reach}(k)$. Then for each neighbor b of r in the intersection graph applies $b \subseteq \textsc{reach}(k+1)$ by Lemma 4.1. This shows that the staged illumination finds the rectangles in the same order as breadth-first search in the intersection graph.

The rectilinear minimum link path problem can thus be solved by forming the intersection graph and searching for the shortest path using breadth-first search [3]. The intersection graph has quadratic size so building it takes at least quadratic time as well. The goal is thus to run the breadth-first search in the intersection graph without explicitly constructing the graph.



(a) Initial state (b) Sweep up (c) Sweep down (d) Select rectangles found by the sweeps (e) Sweep up and down (f) Select rectangles found by the sweeps

Figure 13: The staged illumination proceeds from a starting state (Fig. 13a) by iterating two phases: 1. Illuminate up and down (Fig. 13b, 13c, 13e). 2. Select the illuminated rectangles (Fig. 13d, 13f).

4.2 Staged illumination

We consider again the case where the first link of the path is horizontal. We start by constructing the horizontal decomposition with Algorithm 2.1. We are implicitly searching through the intersection graph of DEC_x and DEC_y, but there is no need to actually compute the vertical decomposition.

$\text{REACH}(0)$ contains the starting point $\mathbf{s}$, and $\text{REACH}(1)$ is the horizontal line segment passing through $\mathbf{s}$. The path finding algorithm works by alternating between two phases [15]:

1. For odd k, $\text{REACH}(k)$ consists of a set of rectangles $R_k \subseteq \text{DEC}_x$ when $k \geqslant 3$. In the special case $k = 1$, $\text{REACH}(k)$ is a horizontal line through a single rectangle of DEC_x. We inspect which rectangles can be reached by a vertical line starting from any point in $\text{REACH}(k)$, and form $\hat{R}_{k+1} = \{r \mid r \in \text{DEC}_x, r \cap \text{REACH}(k+1) \neq \varnothing\}$.
2. For even $k \geqslant 2$ we set $R_{k+1} = \hat{R}_k$. Then $\text{REACH}(k+1) = \bigcup_{r \in R_{k+1}} r$ by Lemma 4.1.

We keep iterating the two phases until the endpoint $\mathbf{t}$ is reached in either phase. Figure 13 illustrates this process.

For performance, it is important to ensure that we don't reilluminate the same parts of the domain too many times. We avoid the reillumination by treating cells of

DEC_x as obstacles if they have been found on some previous illumination step. If a rectangle r is discovered on step k, then it is fully illuminated in $\text{REACH}(k+1)$, and each point visible from r is illuminated in $\text{REACH}(k+2)$. After this there is no longer need to illuminate through r, so we can mark r as an obstacle when 2 steps have passed since the illumination first reached k. This ensures that each rectangle is illuminated only a constant number of times.

Next we show how to implement illumination in y-direction starting from $\text{REACH}(k)$, which is represented as a set of rectangles $R \subseteq \text{DEC}_x$. The illuminated area is computed by two line sweeps, one in direction $+y$ and the other in direction $-y$.

The sweep line stops at each rectangle to be illuminated. During the sweep we maintain the intersection between the sweep line and the newly illuminated region $\text{REACH}(k+1)$. The intersection consists of non-overlapping intervals, which are stored in a binary search tree.

Let $\text{NBS}_d(r)$ and $\text{OBS}_d(r)$ be the neighbor cell and the adjacent obstacles of rectangle r in direction d in DEC_x. Algorithm 4.1 shows the illumination in direction $+y$. The sweep in direction $-y$ is identical.

Lemma 4.2. *The running time of Algorithm 4.1 is $O(m \log m)$ where m is the number of visited rectangles and their neighbor links.*

Proof. The outermost loop processes a different rectangle r each time, so it is executed $O(m)$ times. For each obstacle o adjacent to r, we cut off all the parts of the ranges touching o from the binary search tree. The cutting takes $O(\log m + k)$ time, where k is the number of removed ranges. For each rectangle adjacent to r, we perform a single binary tree operation and possibly a single insertion to the priority queue, which can both be done in $O(\log m)$ time. Thus the total time complexity is $O(m \log m)$. □

The link distance between **s** and **t** can be computed by iteratively performing the illumination sweeps in $+y$ and $-y$ directions until the endpoint is found. Algorithm 4.2 shows the staged illumination process. In the pseudocode we mark the step when the illumination first reaches each rectangle r as `step`$[r]$.

Theorem 3. *Algorithm 4.2 has time complexity $O(n \log n)$ and space complexity $O(n)$.*

Proof. DEC_x is formed in $O(n \log n)$ time by Lemma 2.1. DEC_x has $O(n)$ cells and links between cells. Each cell is turned into an obstacle 2 steps after it has been

Algorithm 4.1 Compute illuminated rectangles in direction $+y$.

Input: A decomposition of the free space into rectangles and a subset of the rectangles R.

Output: Rectangles reachable by moving in direction $+y$ from any point in R.

```
Q ← Priority queue containing the rectangles of R ordered by the topmost
        y-coordinate.
T ← Empty binary search tree.
while Q is not empty do
    r ← pop(Q).
    if r ∈ R then
        Insert x(r) into T.
    end if
    for all o ∈ OBS_{+y}(r) do
        for all t ∈ T touching x(o) do
            Replace t by t\x(o), possibly removing it or splitting it into two parts.
        end for
    end for
    for all b ∈ NBS_{+y}(r) do
        if x(b) touches any range in T and b ∉ Q then
            Insert b into Q.
        end if
    end for
end while
return all the rectangles inserted into Q.
```

Algorithm 4.2 Run staged illumination with the first link horizontal.

Input: Set of obstacle faces O and points $\mathbf{s}$ and $\mathbf{t}$.

Output: Link distance between $\mathbf{s}$ and $\mathbf{t}$.

```
Form DEC_x from O with Algorithm 2.1.
R ← {r ∈ DEC_x | s ∈ r}
if t inside the only rectangle of R then
    return 1 if s_y = t_y, 2 otherwise.
end if
k ← 1
while R ≠ ∅ and t not inside any rectangle of R do
    for all r ∈ R do
        if step[r] is not set then
            step[r] ← k
        else if step[r] ⩽ k − 2 then
            Mark r to be treated as an obstacle during the sweeps.
            Remove r from R.
        end if
    end for
    H_{+y} ← H illuminated in direction +y by Algorithm 4.1.
    H_{−y} ← H illuminated in direction −y by Algorithm 4.1.
    if t found during any of the sweeps then
        return k + 1.
    end if
    H ← H_{+y} ∪ H_{−y}.
    k ← k + 2
end while
return link distance k, or nothing if R = ∅.
```

found. On each iteration of the main loop we perform two sweeps, so each cell is visited at most four times by the sweeps. The time complexity of a single sweep is $O(m \log m)$ by Lemma 4.2, where m is the number of visited rectangles. Each rectangle is visited $O(1)$ times, so the total time taken by all the sweeps is $O(n \log n)$.

The decomposition is created in linear space by Lemma 2.1, and the sweeps only use binary search trees and binary heaps that both take linear space, so the whole algorithm runs in $O(n)$ space. □

The illumination finds the link distance between **s** and the target point **t** by stopping as soon as **t** becomes illuminated. P can become illuminated either during a vertical sweep or the implicit horizontal sweep that happens when we assign $H_{+y} \cup H_{-y}$ into H. The path can then be constructed similarly to regular shortest path computation in a graph: augment the algorithm such that we store into each rectangle how it was initially illuminated, and trace back the path from **t**.

5 Paths in three and higher dimensions

In this section we study the rectilinear minimum link path problem in three-dimensional domains. We present an algorithm with $O(n^2 \log^2 n)$ time and $O(n^2)$ [16] space complexity. We also extend the algorithm to higher dimensions. For any constant $D \geqslant 2$, the presented algorithm works in $O(n^{D-1} \log^{D-1} n)$ time and $O(n^{D-1})$ space.

Some of the basic ideas of the planar minimum link path algorithm can also be applied to the three-dimensional case. The path is computed using the staged illumination paradigm, and on each step we construct $\textsc{reach}(k+1)$ from $\textsc{reach}(k)$ by applying a sweep plane algorithm in all the coordinate axis directions. However in the three-dimensional case the representation of the illuminated region is more complicated, and the sweep algorithm requires more sophisticated data structures.

We use the decomposition presented in Section 2.2 to maintain the illuminated region and to guide the illumination during the sweeps. Unlike the planar case, the algorithm does not depend on any specific decomposition but can be used with any space decomposition. The running time of the algorithm depends on the number of cells and links between them, and the selected decomposition has the advantage of having a small number of both of them.

The decomposition is used to implement the plane sweep algorithm for computing

REACH$(k+1)$ from REACH(k). On each step we run six sweeps, one in each of the directions $\pm x$, $\pm y$ and $\pm z$. Combination of the results of the sweeps in each direction gives the whole region REACH$(k+1)$. Next we discuss how the sweep plane algorithm is implemented, and then show how the minimum link path algorithm is built on top of it.

5.1 Illumination by plane sweep

We use a sweep plane algorithm to compute REACH$(k+1)$ based on REACH(k) [16]. The new region is computed by applying the sweep in each coordinate axis direction $\pm x$, $\pm y$ and $\pm z$. The sweep performs two functions: it discovers which cells can be illuminated from REACH(k), and it constructs the boundary of the illuminated region. The boundary is used on the next step to compute REACH$(k+2)$ from REACH$(k+1)$. Note that while the region REACH(k) plays a central role in the design of the algorithm, we don't explicitly maintain this region.

During each of the six sweeps we maintain the intersection of the sweep plane with the newly illuminated region, and a priority queue of events. Each event belongs to one of the following types:

`AddRectangleEvent` where a previously illuminated region is added to the sweep plane.

`CellEvent` which occurs when the sweep plane reaches the end of a cell that potentially intersects with the illuminated region.

`ObstacleEvent` which occurs when the sweep plane encounters an obstacle face with normal opposite to the sweep direction.

Consider a sweep in direction $+z$. In a `CellEvent` we first check whether any part of the reached cell c is illuminated. If the cell is not illuminated, we discard the event. Otherwise we create a `CellEvent` for each neighbor of c in direction $+z$, and an `ObstacleEvent` for each obstacle adjacent to c in direction $+z$. In an `ObstacleEvent` the projection of the encountered obstacle is cleared from the sweep plane. Additionally the cleared rectangles are used to construct new `AddRectangleEvent`s on the boundary of the illuminated region to be used on the next illumination step.

During the sweep, the intersection of the illuminated region with the sweep plane is stored in a two-dimensional unified segment tree. Each `AddRectangleEvent` inserts

a new rectangle to the tree. Each illuminated canonical node of the tree stores a reference to the event that was used to generate it. If multiple `AddRectangleEvent`s illuminate the same canonical node, only the one inserted first is stored. This reference is used to determine the z-range of the `AddRectangleEvent`s generated during the `ObstacleEvent`s, as well as for tracing back the path from $\mathbf{t}$ to $\mathbf{s}$ when the illumination finishes. Algorithm 5.1 describes the sweep in $+z$ direction. The sweeps in the other directions are identical.

Algorithm 5.2 implements the staged illumination method by iteratively applying the sweep algorithm. We can find the minimum link path between two points by stopping the illumination when it reaches the endpoint and tracing back the path. The stopping condition and path generation have been omitted to focus on the core algorithm.

5.2 Event generation

The sweep operation requires an initial set of events as an input. This set should contain `AddRectangleEvent`s as well as `CellEvent`s for all the cells touched by the rectangles of the `AddRectangleEvent`s. We create `CellEvent`s for all the cells illuminated during the previous step. This may create some unnecessary `CellEvent`s, but they don't hinder the sweep algorithm or affects its asymptotic running time. The `AddRectangleEvent`s for the next illumination step are generated during the processing of `ObstacleEvent`s. We explain again only the $+z$ sweep, the other directions being identical. The goal is to generate `AddRectangleEvent`s on the boundary of the space illuminated during illumination of $\textsc{reach}(k)$ such that the sweeps on the following step correctly generate $\textsc{reach}(k+1)$.

During a $+z$ sweep we generate `AddRectangleEvent`s in directions $\pm x$ and $\pm y$. On an `ObstacleEvent` we perform a clear operation on the unified tree containing the sweep plane status. This clears some of the tree nodes. As a first step in the event generation, we create an `AddRectangleEvent` on each side of every cleared canonical node.

Each generated event e is assigned a rectangle $\texttt{rect}(e)$ that will be added to the unified tree when the event is processed. Each rectangle is defined by two intervals; for example if e is an event in direction $+x$ then $\texttt{rect}(e)$ is defined by a z-range and a y-range. The z-range is bounded by the z-coordinate of the `ObstacleEvent` generating e and the z-coordinate of the `AddRectangleEvent` that added the cleared

Algorithm 5.1 Illuminate by a plane sweep in direction $+z$ starting from provided events. Produces `AddRectangleEvent`s and `CellEvent`s in directions $\pm x$ and $\pm y$ for the next illumination step.

Input: Initial event set E.

Output: List of `AddRectangleEvent`s and `CellEvent`s for the next illumination step.

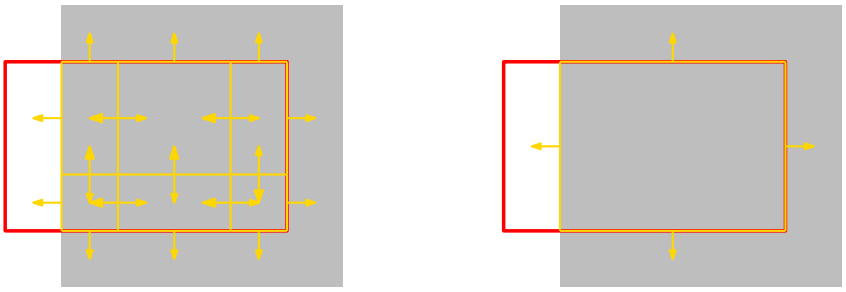
```
Q ← Priority queue containing E.
T ← Empty two-dimensional unified segment tree.
R ← Empty list of output events.
while Q is not empty do
    e ← pop(Q).
    if e is AddRectangleEvent then
        Insert rect(e) into T.
    else if e is CellEvent then
        if rect(e) touches any rectangle in T then
            c ← cell(e).
            Insert CellEvents for c in directions ±x and ±y into R.
            for all o ∈ OBS_{+z}(c) do
                Insert ObstacleEvent for o into Q.
            end for
            for all b ∈ NBS_{+z}(c) do
                Insert CellEvent for b into Q.
            end for
        end if
    else if e is ObstacleEvent then
        Clear rect(e) from T.
        Generate AddRectangleEvents for the cleared canonical nodes into R
            (Section 5.2).
    end if
end while
return R.
```

Algorithm 5.2 Run staged illumination in a three-dimensional domain.

Input: Set of obstacle faces and a starting point **s**.

Effect: Discovers all the points reachable from **s** with the staged illumination.

```
Compute decomposition of A into cells using Algorithm 2.2.
for all d ∈ {±x, ±y, ±z} do
    E_d ← Event set containing an AddRectangleEvent at s with width and
          height 0 and a CellEvent for the cell containing s.
end for
while E_d is not empty for some d do
    ▷ E_d-sets define the boundary of REACH(k).
    for all d ∈ {±x, ±y, ±z} do
        E'_d ← Empty event set.
    end for
    for all d ∈ {±x, ±y, ±z} do
        Illuminate in direction d using Algorithm 5.1 with events E_d.
        Store the returned events into E'_e for e ≠ ±d.
    end for
    for all d ∈ {±x, ±y, ±z} do
        E_d ← E'_d.
    end for
end while
```



(a) Initially generated `AddRectangleEvent`s

(b) `AddRectangleEvent`s after the filtering

Figure 14: Initially we can create a large number of `AddRectangleEvent`s because of the segment tree structure, but the filtering clears the redundant ones.

canonical node. The other range is defined by the side of the cleared canonical node.

Creating `AddRectangleEvent`s for each cleared canonical node can lead to a large number of events, because each rectangle is split into $O(\log^2 n)$ canonical nodes in the tree. To reduce the number of generated events we perform *filtering*, which consists of the following two steps:

1. Remove all the pairs of `AddRectangleEvent`s with the same position but opposite direction.
2. Merge all sequences of aligned `AddRectangleEvent`s into a single `AddRectangleEvent`.

The aligned `AddRectangleEvent`s in direction $+x$ are the ones that occur on the same x coordinate and have the same z-ranges and the y-ranges have a common endpoint. The definition is similar in the other directions. The aligned `AddRectangleEvent`s are searched for the events in all the directions separately. The filtering allows us to bound the number of `AddRectangleEvent`s generated during the illumination. Figure 14 shows the generated `AddRectangleEvent`s on an obstacle before and after the filtering.

Filtering can be done in linear time with respect to the number of unfiltered rectangles, provided that we have preallocated an array of $O(n^2)$ empty lists. To see why, notice that there are only $O(n^2)$ possible coordinates that a corner of a rectangle can have. We can group the rectangles by their corners into an array of size $O(n^2)$ and use the array to find matching pairs of rectangles to remove or merge.

To make the minimum link path computation efficient, we want to avoid reilluminating

the same cells multiple times. In the algorithm for the planar case this was done by turning cells into obstacles once enough steps have passed since the cell was first discovered. This approach works also for the three-dimensional case, but since there are $O(n^2)$ cells and clearing a rectangle from the unified tree requires $O(n \log n)$ steps, generating `ObstacleEvent`s for each of the cells would increase the time complexity significantly.

To overcome this problem, instead of limiting the reachable cells, we limit the creation of `AddRectangleEvent`s. Algorithm 5.1 creates `AddRectangleEvent`s for the next step when the sweep plane hits an obstacle. If the illumination first reaches an obstacle face o on step k, then after step $k+2$ all the points touching o are illuminated, so after step $k+3$ all the points visible from o are illuminated. Thus we can stop creating `AddRectangleEvent`s for an obstacle o that was first found more than 3 steps ago, as any point illuminated through such events has been discovered already.

5.3 Complexity

We now prove the time and space complexity of the minimum link path algorithm [16].

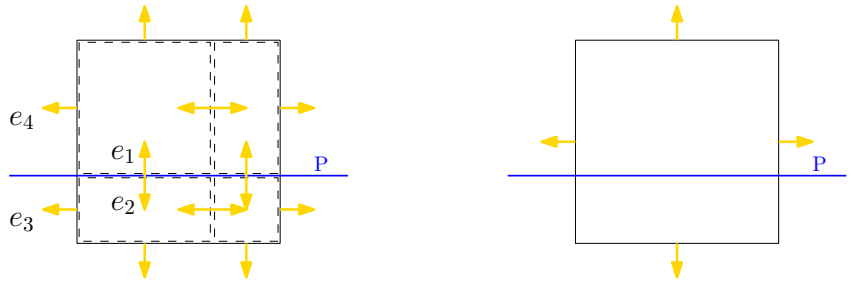
Lemma 5.1. *For each obstacle we generate an* `ObstacleEvent` *on at most 9 steps of the illumination.*

Proof. Recall that we stop generating `AddRectangleEvent`s on an obstacle when 3 steps have passed since the obstacle was discovered. Suppose that we first generate `ObstacleEvent` for an obstacle o on step k. After step $k+3$ all the points visible from o are illuminated. After step $k+4$ we have discovered each obstacle h such that the illumination reaching h could generate an `AddRectangleEvent` that illuminates o. We won't generate any `AddRectangleEvent`s for such obstacle h after step $k+7$, so the last time a sweep can reach o is on step $k+8$. □

Lemma 5.2. *For each cell we generate a* `CellEvent` *on at most 9 steps of the illumination.*

Proof. The proof is very similar to Lemma 5.1. If we discover a cell c on step k then all the points visible from c are illuminated after step $k+3$, so no more sweeps can reach c after step $k+8$. □

Lemma 5.3. *The number of* `AddRectangleEvent`*s generated during the illumination after the filtering is* $O(n^2)$.



(a) Initially generated `AddRectangleEvents`

(b) `AddRectangleEvents` after the filtering

Figure 15: We may initially generate `AddRectangleEvents` on a plane P because of how the rectangles are split to canonical nodes in the unified segment tree. In Figure 15a an event is on the plane P, but it is removed together with e_2. Similarly illumination from e_3 might create events on P on the next step, but we combine it with e_4 to a larger `AddRectangleEvent` that no longer has boundary on P.

Proof. We only count the `AddRectangleEvents` in direction $+z$. The other directions are identical, and there is only a constant number of directions. Each `AddRectangleEvent` lies on a common plane with some obstacle face whose normal points to direction $-z$. For the purpose of deriving an upper bound, we assume that each obstacle face has a unique z-coordinate. We may overcount the number of `AddRectangleEvents` this way by counting the some events multiple times, but we only need to prove an upper bound.

Consider the `ObstacleEvents` of an obstacle o on a plane P. Events in direction $+z$ can be generated during the sweeps to direction $\pm x$ and $\pm y$. We only consider the sweep in direction $+y$. We generate an event on the plane P if an `ObstacleEvent` removes a rectangle R from the unified tree such that the z-ranges of R is bounded by P.

Some `AddRectangleEvents` might be generated on P because of how the rectangles are stored in the unified segment tree. When a rectangle is added to the tree it is split to canonical nodes, and some of them might have boundary on P even if the original rectangle didn't. In such cases we generate `AddRectangleEvents` from canonical nodes on both sides of P, so the events are removed during the filtering where we remove pairs of events at identical positions with opposite directions. Figure 15a shows an example of such a situation.

We may also generate `AddRectangleEvents` in directions $\pm x$ with z-range bounded by P because the z-range of some canonical nodes can be bounded by P. In the

second part of the filtering we merge aligned events into larger events. Because we generate the events on both sides of P, we will merge them so that the z-coordinate of P is strictly inside z-range of the combined event. This is illustrated by the events in horizontal directions in Figure 15.

Because of the filtering, we can only have events on the plane P after the sweep has reached the obstacle o. There are three cases for how an event might be generated on the plane P that is not removed by the filtering.

1. During a sweep in direction $+z$ we hit the obstacle o. We generate `AddRectangleEvent`s in directions $\pm x$ and $\pm y$ that are bounded by P. During the sweeps of the following step we add the rectangles of these events to the unified tree. When these sweeps hit an `ObstacleEvent`, we may then generate events in direction $+z$ on the plane P.

2. During a sweep in direction $\pm x$ or $\pm y$ the obstacle face o cuts a rectangle in the tree such that the remaining part has a boundary on P. Then when the sweep processes another `ObstacleEvent`, we generate an `AddRectangleEvent` in direction $+z$ on the plane P.

3. If on step k we created `AddRectangleEvent` for step $k + 1$ on the plane P, then we also created events in directions $\pm x$ or $\pm y$ whose z-ranges are bounded by P. Those rectangles are added to the unified tree during a sweep of step $k + 1$. We then create `AddRectangleEvent`s on P for step $k + 2$ when the sweep processes an `ObstacleEvent`.

First let us consider the first two types of events. They only occur through sweeps touching the surface of obstacle o. By Lemma 5.1 and Lemma 5.2 such sweeps occur only on $O(1)$ steps. On a single sweep, the number of events created on P is bounded by the number of canonical nodes that have P as one of their sides. There are $O(n)$ such nodes, so the number of events is $O(n)$.

The following observation allows us to bound the number of type 3 events: Let $\mathcal{B}$ be the cross section of the free space $\mathcal{A}$ with the plane P. After the illumination has reached the obstacle o, the illumination through the events bounded on one side by P proceeds exactly like the two-dimensional staged illumination in $\mathcal{B}$. The two-dimensional illumination illuminates only maximal rectangles of DEC_x and DEC_y, producing a total of $O(n)$ illuminated rectangles according to Theorem 3. There is some overhead because we may reilluminate the same parts multiple times, but

by Lemma 5.2 each part of the domain can be illuminated only $O(1)$ times. Thus the number of type 3 events is within a constant factor of the event count of the two-dimensional staged illumination.

The number of events on the plane of each obstacle face is $O(n)$, so the total number of events is $O(n^2)$. □

We are ready to prove the main result of this section.

Theorem 4. *A rectilinear minimum link path in a three-dimensional rectilinear domain can be computed in $O(n^2 \log^2 n)$ time and $O(n^2)$ space.*

Proof. A minimum link path can be computed by running the staged illumination of Algorithm 5.2 and tracing back the path when the illumination reaches the endpoint.

By Lemma 5.2 each cell is visited $O(1)$ times during the illumination. On each `CellEvent` we perform a single lookup to the unified tree and iterate over the neighbors of the cell. By Lemma 3.9 the lookup can be done in $O(\log^2 n)$ time. Since the decomposition has $O(n^2)$ cells and $O(n^2)$ links, the total time taken in processing `CellEvent`s is $O(n^2 \log^2 n)$.

On each `AddRectangleEvent` we insert a new rectangle to the unified tree in $O(\log^2 n)$ time by Lemma 3.9. By Lemma 5.3 the number of `AddRectangleEvent`s is $O(n^2)$, so the events are processed in $O(n^2 \log^2 n)$ time. The filtering of `AddRectangleEvent`s can be done in linear time, and the total number of unfiltered events is at most the number of `AddRectangleEvent`s multiplied by the $O(\log^2 n)$ canonical nodes used to store a rectangle in the unified tree. Thus the time spent in the filtering is also $O(n^2 \log^2 n)$.

On each `ObstacleEvent` we clear a rectangle from the tree. The clearing takes $O(n \log n + k)$ time, where k is the number of cleared canonical nodes. Each `AddRectangleEvent` adds a rectangle to $O(\log^2 n)$ canonical nodes, so the total number of nodes to be cleared is $O(n^2 \log^2 n)$. There are $O(n)$ `ObstacleEvent`s by Lemma 5.1, so the total time taken in the `ObstacleEvent`s is $O(n^2 \log^2 n)$.

As all parts of the algorithm are done in $O(n^2 \log^2 n)$ time, the total time complexity is $O(n^2 \log^2 n)$.

For the space complexity, we need to consider the maximal number of events that need to be stored in the memory. The number of `AddRectangleEvent`s after the filtering is $O(n^2)$, but the number before filtering is $O(n^2 \log^2 n)$ as each `AddRectangleEvent` adds $O(\log^2 n)$ rectangles to the unified tree. A single `ObstacleEvent` may create at

most $O(n^2)$ `AddRectangleEvent`s because the unified tree has $O(n^2)$ nodes, so we never need to store more than $O(n^2)$ `AddRectangleEvent`s in the memory. The size of the decomposition as well as the unified segment tree used during the sweeps is also $O(n^2)$, so the space complexity is $O(n^2)$. □

5.4 Higher dimensional paths

The minimum link path algorithm for three-dimensional domains can be generalized to higher dimensions. We now assume that $\mathcal{A}$ is a D-dimensional rectilinear region for an arbitrary constant $D \geqslant 2$. We develop a generalization of Algorithm 5.2 that computes a rectilinear minimum link path in a D-dimensional domain in $O(n^{D-1} \log^{D-1} n)$ time and $O(n^{D-1})$ space.

The illumination in the D-dimensional space is done using the same kinds of events as were used in the three-dimensional case. The sweep plane is replaced by a $(D-1)$-dimensional hyperplane, and each event e defines a $(D-1)$-dimensional hyperrectangle `rect`(e).

We use a $(D-1)$-dimensional unified segment tree to store the sweep hyperplane state during the sweeps. The unified tree and the D-dimensional space decomposition allow us to use an almost identical algorithm for the high dimensional case as the algorithm for three-dimensional domains. Algorithm 5.3 shows the sweep in direction $+D$, the other directions being identical. Algorithm 5.4 implements the D-dimensional staged illumination by performing the hyperplane sweep in all the directions on each step. We omit again the final path generation from the pseudocode.

We use the same method for avoiding reilluminating as in the three-dimensional case. For each obstacle o we record when it was first discovered, and only generate `AddRectangleEvent`s from o until a certain number of steps has passed. If o is reached first on step k, then at the end of step $k + D$ all the points visible from o are illuminated, so we can stop creating `AddRectangleEvent`s from o starting from step $k + D + 1$.

We also need to perform the filtering described in Section 5.2 to avoid generating a large number of `AddRectangleEvent`s because of the unified tree structure. The filtering rules are the same as the ones used in the three-dimensional algorithm, except that we are now working with $(D-1)$-dimensional hyperrectangles. The operations below are performed for the `AddRectangleEvent`s produced by the sweep in direction $+D$.

Algorithm 5.3 Illuminate by a hyperplane sweep in direction $+D$ starting from provided events. Produces `AddRectangleEvent`s and `CellEvent`s in directions other than $\pm D$ for the next illumination step.

Input: Initial event set E.

Output: List of `AddRectangleEvent`s for the next illumination step.

$Q \leftarrow$ Priority queue containing E.
$T \leftarrow$ Empty $(D-1)$-dimensional unified segment tree.
$R \leftarrow$ Empty list of output events.
while Q is not empty **do**
  $e \leftarrow \texttt{pop}(Q)$.
  if e is `AddRectangleEvent` **then**
    Insert $\texttt{rect}(e)$ into T.
  else if e is `CellEvent` **then**
    if $\texttt{rect}(e)$ touches any hyperrectangle in T **then**
      $c \leftarrow \texttt{cell}(e)$.
      Insert `CellEvent`s for c in directions other than $\pm D$ into R.
      for all $o \in \mathrm{OBS}_{+D}(c)$ **do**
        Insert `ObstacleEvent` for o into Q.
      end for
      for all $b \in \mathrm{NBS}_{+D}(c)$ **do**
        Insert `CellEvent` for b into Q.
      end for
    end if
  else if e is `ObstacleEvent` **then**
    Clear $\texttt{rect}(e)$ from T.
    Generate `AddRectangleEvent`s for the cleared canonical hyperrectangles into R.
  end if
end while
return R.

Algorithm 5.4 Run staged illumination in a D-dimensional domain.

Input: Set of obstacle faces and a point $\mathbf{s}$.

Effect: Discovers all the points reachable from $\mathbf{s}$ with the staged illumination.

```
Compute decomposition of A into D-dimensional cells using Algorithm 2.4.
for all d ∈ {±1, ..., ±D} do
    E_d ← Event set containing an AddRectangleEvent at s with size 0 and a
          CellEvent for the cell containing s.
end for
while E_d is not empty for some d do
    for all d ∈ {±1, ..., ±D} do
        E'_d ← Empty event set.
    end for
    for all d ∈ {±1, ..., ±D} do
        Illuminate in direction d using Algorithm 5.3 with events E_d.
        Store the returned events into E'_e for e ≠ ±d.
    end for
    for all d ∈ {±1, ..., ±D} do
        E_d ← E'_d.
    end for
end while
```

1. Remove all pairs of `AddRectangleEvent`s with the same position but opposite direction.

2. Merge all sequences of aligned `AddRectangleEvent`s into a single `AddRectangleEvent`.

Two `AddRectangleEvent`s are aligned if they are identical in all but a single direction d, and their ranges in direction d are next to each other. We check for aligned rectangles in each direction d separately. The filtering process can again be done in linear time by indexing the rectangles by their corners into an array of size $O(n^{D-1})$.

Next we prove the running time of the algorithm.

Lemma 5.4. *For each obstacle we generate an* `ObstacleEvent` *on at most* $2D + 3$ *steps of the illumination.*

Proof. The proof is similar to Lemma 5.1 for the three-dimensional case. We stop generating `AddRectangleEvent`s from an obstacle when D steps have passed since that obstacle was discovered. Suppose the illumination first reaches an obstacle o on step k. After step $k + D$ all the points visible from o are illuminated. After step $k + D + 1$ we have discovered each obstacle h such that the illumination reaching h could generate an `AddRectangleEvent` that illuminates o. We won't generate any `AddRectangleEvent`s for such obstacle h after step $k + 2D + 1$, so the last time a sweep can reach o is on step $k + 2D + 2$. □

Lemma 5.5. *For each cell we generate a* `CellEvent` *on at most* $2D + 3$ *steps of the illumination.*

Proof. The proof is very similar to Lemma 5.4. If we discover a cell c on step k then all the points visible from c are illuminated after step $k + D$, so no more sweeps can reach c after step $k + 2D + 2$. □

Lemma 5.6. *The number of* `AddRectangleEvent`*s generated during the illumination after the filtering is* $O(n^{D-1})$.

Proof. The proof is similar to Lemma 5.3. We only count the `AddRectangleEvent`s in direction $+D$ occurring on a hyperplane P defined by a $(D-1)$-dimensional obstacle face o. To simplify the counting we again assume that the obstacle D-coordinates are unique, so o is the only obstacle face on P. We show that the number of events generated on P is $O(n^{D-2})$, which proves the claim.

We classify the events remaining after the filtering similarly to the three-dimensional case:

1. During a sweep in direction $+D$ we hit the obstacle o. We generate `AddRectangleEvent`s in directions other than $\pm D$ that are bounded by P. During the sweeps of the following step we add the rectangles of these events to the unified tree. When these sweeps hit an `ObstacleEvent`, we may then generate events in direction $+D$ on the plane P.

2. During a sweep in a direction other than $\pm D$ the obstacle face o cuts a hyperrectangle in the tree such that the remaining part has a boundary on P. Then when the sweep processes another `ObstacleEvent`, we generate an `AddRectangleEvent` in direction $+D$ on the plane P.

3. If on step k we created `AddRectangleEvent` on plane P, then we also created events in directions other than $\pm D$ whose D-ranges are bounded by P. Those rectangles are added to the unified tree during a sweep of step $k + 1$. We then create `AddRectangleEvent`s on P for step $k + 2$ when the sweep processes an `ObstacleEvent`.

Similar to the three-dimensional case, a single sweep can only produce as many events on P as the number of canonical nodes whose hyperrectangles are bounded by P. In a $(D-1)$-dimensional unified tree the number of such nodes is $O(n^{D-2})$. By Lemma 5.4 and Lemma 5.5 the illumination can only touch o on $O(1)$ steps, so the number of type 1 and 2 events on P is $O(n^{D-2})$.

We use induction on D to prove the number of type 3 events. In the base case $D = 2$ the obstacle o is a line segment and P is the line passing through the segment. In this case events on P can only be generated if the sweep touches o. This happens only $O(1)$ times so the number of events on P is $O(1)$.

For $D \geqslant 3$ let $\mathcal{B}$ be the cross section of the free space $\mathcal{A}$ with the plane P. We notice again that the illumination through the events bounded by P advances exactly like the $(D-1)$-dimensional illumination in $\mathcal{B}$. By induction the $(D-1)$-dimensional illumination creates $O(n^{D-2})$ events. Multiplying this by the number of times any cell can be visited gives an upper bound for the D-dimensional illumination on P. Since there is only a constant factor overhead compared to the $(D-1)$-dimensional illumination, the number of type 3 events on P is $O(n^{D-2})$.

The number of events on the hyperplane of each obstacle is $O(n^{D-2})$, so the total number of events is $O(n^{D-1})$. □

We are now ready to state the final result.

Theorem 5. *A rectilinear minimum link path in a D-dimensional rectilinear domain can be computed in $O(n^{D-1} \log^{D-1} n)$ time and $O(n^{D-1})$ space for any constant $D \geqslant 2$.*

Proof. A minimum link path can be computed by running the staged illumination of Algorithm 5.4 and tracing back the path when the illumination reaches the endpoint.

By Lemma 5.5 each cell is visited $O(1)$ times during the illumination. On each `CellEvent` we perform a single lookup to the unified tree and iterate over the neighbors of the cell. By Lemma 3.9 the lookup can be done in $O(\log^{D-1} n)$ time. Since the decomposition has $O(n^{D-1})$ cells and $O(n^{D-1})$ links, the total time taken in processing `CellEvent`s is $O(n^{D-1} \log^{D-1} n)$.

On each `AddRectangleEvent` we insert a new rectangle to the unified tree in $O(\log^{D-1} n)$ time by Lemma 3.9. By Lemma 5.6 the number of `AddRectangleEvent`s is $O(n^{D-1})$, so the events are processed in $O(n^{D-1} \log^{D-1} n)$ time. The filtering of `AddRectangleEvent`s can be done in linear time, and the total number of unfiltered events is at most the number of `AddRectangleEvent`s multiplied by the $O(\log^{D-1} n)$ canonical nodes used to store a hyperrectangle in the unified tree. Thus the time spent in filtering is also $O(n^{D-1} \log^{D-1} n)$.

On each `ObstacleEvent` we clear a hyperrectangle from the tree. By Lemma 3.10 each clear operation takes $O(n^{D-2} \log n + k)$ time, where k is the number of cleared canonical nodes. Each `AddRectangleEvent` adds a rectangle to $O(\log^{D-1} n)$ canonical nodes, so the total number of nodes to be cleared is $O(n^{D-1} \log^{D-1} n)$. There are $O(n)$ `ObstacleEvent`s by Lemma 5.4, so the total time taken in `ObstacleEvent`s is $O(n^{D-1} \log^{D-1} n)$.

All parts of the algorithm are done in $O(n^{D-1} \log^{D-1} n)$ time, so the total time complexity is $O(n^{D-1} \log^{D-1} n)$.

For the space complexity notice that we may create at most as many `AddRectangleEvent`s during a single `ObstacleEvent` as there are nodes in the unified tree, so only $O(n^{D-1})$ events need to be stored in the memory. The decomposition and the unified segment tree both require $O(n^{D-1})$ space, so the space complexity is $O(n^{D-1})$. □

6 Conclusions

We studied the rectilinear minimum link path problem in two, three and higher dimensions. We presented new algorithms for the problem with lower asymptotic running times than the previously known solutions. The results are interesting since computing rectilinear minimum link paths is an old and well studied problem [16, 21].

We gave a simple algorithm for the planar rectilinear minimum link path problem with running time $O(n \log n)$ [15]. The algorithm has the same asymptotic complexity as the older results [3, 17], but it uses simpler data structures. We also provided a new algorithm for the three-dimensional variant of the problem with running time $O(n^2 \log^2 n)$ [16]. Finally, we showed that the algorithm can be generalized to work in any dimension D with running time $O(n^{D-1} \log^{D-1} n)$. The new algorithms provide significant improvements over the previous best known results $O(n^{2.5} \log n)$ [21] and $O(n^D \log n)$ [6].

The key techniques for achieving the new results were the recursive space decomposition (Section 2) and the multidimensional segment tree (Section 3). The algorithms are relatively simple, which is demonstrated by the implementation linked in Appendix 1.

A natural follow up question is whether the results could be improved further. The planar problem was shown to be as difficult as sorting the input [3], so the known algorithms are asymptotically optimal. For the higher dimensional versions no other lower bound is known besides the $O(n \log n)$ bound. The planar minimum link path problem without orientation restriction was shown to be 3SUM-hard by Mitchell, Polishchuk and Sysikaski [15], but there is no known way to extend this result to the rectilinear case.

One of the main bottlenecks of the D-dimensional minimum link path algorithm are operations on the unified segment tree. A potential way to speed up the algorithm would be to develop a faster data structure for hyperrectangle intersection testing. Fractional cascading is a technique that allows speeding up operations in some D-dimensional data structures from $O(\log^D n)$ to $O(\log^{D-1} n)$ [1, 12]. Wagner [20] suggests that this technique might be applicable also for the unified segment tree. However the idea has not been explored further, so developing a faster multidimensional segment tree variant remains as a future challenge.

References

[1] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[3] Gautam Das and Giri Narasimhan. Geometric searching and link distance. In *Proceedings of the Workshop on Algorithms and Data Structures*, pages 261–272. LNCS 519, Springer, 1991.

[4] Mark De Berg. On rectilinear link distance. *Computational Geometry*, 1(1):13–34, 1991.

[5] Mark De Berg, Otfried Cheong, Marc Var Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[6] Mark De Berg, Marc Van Kreveld, Bengt J. Nilsson, and Mark Overmars. Shortest path queries in rectilinear worlds. *International Journal of Computational Geometry & Applications*, 2(3):287–309, 1992.

[7] Adrian Dumitrescu, Joseph S. B. Mitchell, and Micha Sharir. Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. *Discrete & Computational Geometry*, 31(2):207–227, 2004.

[8] Robert Fitch, Zack Butler, and Daniela Rus. 3D rectilinear motion planning with minimum bend paths. In *Proceedings of the International Conference on Intelligent Robots and Systems*, volume 3, pages 1491–1498. IEEE, 2001.

[9] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[10] Hiroshi Imai and Takao Asano. Efficient algorithms for geometric graph search problems. *SIAM Journal on Computing*, 15(2):478–494, 1986.

[11] Antti Laaksonen. *Guide to Competitive Programming*. Springer, 2018.

[12] George S. Lueker. A data structure for orthogonal range queries. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 28–34. IEEE, 1978.

[13] Koichi Mikami and Kinya Tabuchi. A computer program for optimal routing of printed circuit connectors. *Proceedings of the International Federation for Information Processing Congress*, 1968.

[14] Joseph S. B. Mitchell. Geometric shortest paths and network optimization. *Handbook of Computational Geometry*, 334:633–702, 2000.

[15] Joseph S. B. Mitchell, Valentin Polishchuk, and Mikko Sysikaski. Minimum-link paths revisited. *Computational Geometry*, 47(6):651–667, 2014.

[16] Valentin Polishchuk and Mikko Sysikaski. Faster algorithms for minimum-link paths with restricted orientations. In *Proceedings of the Workshop on Algorithms and Data Structures*, pages 655–666. LNCS 3317, Springer, 2011.

[17] Masao Sato, Jiro Sakanaka, and Tatsuo Ohtsuki. A fast line-search method based on a tile plane. In *Proceedings of the International Symposium on Circuits & Systems*, pages 588–591. IEEE, 1987.

[18] Vijay K. Vaishnavi. Computing point enclosures. *IEEE Transactions on Computers*, 31(1):22–29, 1982.

[19] David P. Wagner. *Path planning algorithms under the link-distance metric*. PhD thesis, Dartmouth College, Hanover, New Hampshire, 2006.

[20] David P. Wagner. The unified segment tree and its application to the rectangle intersection problem. In *Proceedings of the 25th Canadian Conference on Computational Geometry*, 2013.

[21] David P. Wagner, Robert Scot Drysdale, and Clifford Stein. An $O(n^{5/2} \log n)$ algorithm for the rectilinear minimum link-distance problem in three dimensions. *Computational Geometry: Theory and Applications*, 42:376–387, 2009.

Appendix 1. Source code location

An implementation of the D-dimensional rectilinear minimum link path algorithm presented in the thesis is available in the following URL:
`https://github.com/sisu/gradu/tree/master/code`

The code implements the minimum link path computation in time $O(n^{D-1} \log^{D-1} n)$ for any constant D defined at compile time. The code is written in C++ using the C++14 standard. The implementation does not depend on any libraries besides the C++ standard library, but the associated unit tests are written using the Google Test framework.